DYNAMIC METHOD INVOCATION 67 | INSIDE JAVA CARD 77 | JAVA QUIZ 91

Java™ magazine

By and for the Java community

NOVEMBER/DECEMBER 2017

JAVA EE 8

WHAT YOU NEED TO KNOW

13 SERVLET 4.0 | 23 CDI 2.0 | 43 JPA 2.2 | 56 MICROPROFILE

ORACLE.COM/JAVAMAGAZINE

ORACLE

//table of contents /

COVER FEATURES

OTHER FEATURES

DEPARTMENTS

COVER ART BY WES ROWELL

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editors
Karen Perkins, Jim Donahue

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Publication Designer
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher and Audience Development Director
Karin Kinnear

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa

Account Manager
Mark Makinney

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@omeda.com

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by
Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

iText Developers Platform

Build on proven technology

The iText Developers Platform is a revenue sharing partnership that takes your creative iText 7 add-on application and leverages our experience in sales and marketing, by selling the product as part of the iText 7 Suite.

Interested?

Here is what it takes:

Idea → Build POC → Apply → Create → Go to Market

Learn more at: www.itextpdf.com/itext-developer-platform

Java EE Development Goes Open Source

Development of future releases will be hosted at the Eclipse Foundation.

In an unexpected and widely applauded move, Oracle announced just before the JavaOne conference this year that it would be moving development of Java EE to the open source community. This action, effectively unthinkable a few years ago, is being done by giving control of development technologies and of project governance to the Eclipse Foundation. Included in this transition are the full source code of the different reference implementations and of the many test suites that ensure conformance and compliance with Java EE specification requirements.

This migration shows emphatically that Oracle is giving the technology to the community. That is, this move should not be confused with the occasional dumping of technologies to open source foundations by companies no longer interested in supporting them—a phenomenon known as "abandon-ware." Rather, Oracle has pledged to move its commercial Java EE offering, Oracle WebLogic, to conformance with Java EE 8 and to continue contributing to the development and evolution of the standards, along with vendors Red Hat, IBM, Tomitribe, and Payara, as well as the large contributing community of developers.

The Eclipse Foundation was chosen to be the host due in part to its previous experience hosting Java EE technologies, such as JPA and JSON-B. In addition, it currently hosts the complementary MicroProfile project, which is examined in detail on page 56 in this issue. Conversations with Eclipse officials revealed that they expect the transition to take approximately a year. Why so long? Not only do dozens of codebases and supporting documents need to be migrated to Eclipse servers, but a substantial amount of policy needs

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

to be formulated, such as staffing the individual projects—who gets to commit, who reviews changes, and who runs the projects—as well as larger questions such as how will conformance with Java EE standards be validated, what will be the process for determining a new release, and so on. As David Delabassée of Oracle points out, there are additional issues to resolve as well, such as branding and the possible integration of MicroProfile.

With regard to branding, it got off to a bit of a rocky start at JavaOne when rumor had it that Java EE would be renamed EE4J (Eclipse Enterprise for Java). However, that is actually the proposed name of the project at the Eclipse Foundation, rather than of the technology itself. The relationship between EE4J and Java EE is analogous to OpenJDK and Java SE—the former is the development project, and the latter is the resulting technology.

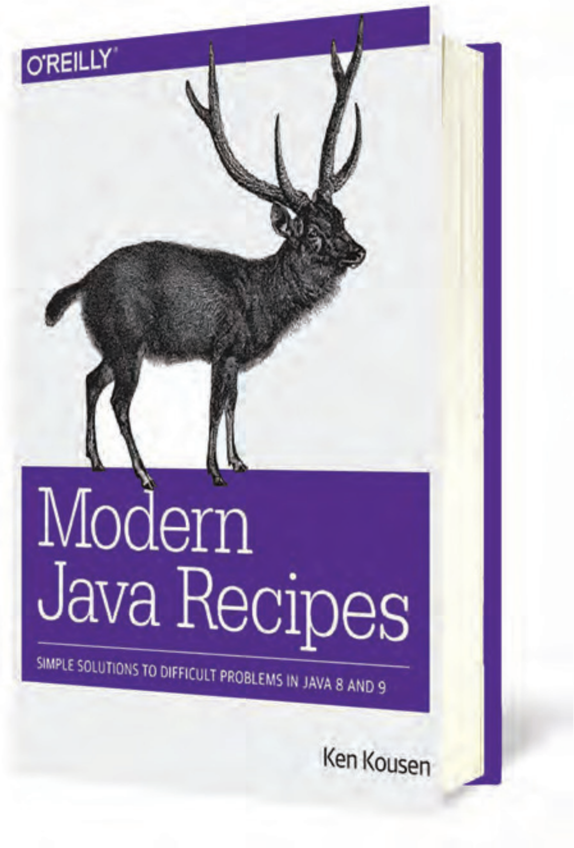
Community reaction to the move by Oracle has been uniformly supportive. And there's good reason for that enthusiasm. Unlike many other projects transferred to open source, Java EE benefits from a very active community that continues to push forward the multiple constituent technologies. For example, in this issue, we look at how those communities, along with Oracle, have significantly updated CDI, Servlet, and JPA. But certainly, we could have included other technologies, too—many of which are driven by active expert groups donating their time and effort to the project.

This strong community, more than any other aspect, I believe, guarantees the success of this migration. If all goes well, as I expect it will, the migration should make it possible to attract even more developers to grow and advance these technologies.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

P.S. The events described here are under active implementation and discussion; so it's entirely possible that details of the transition and of the project at the Eclipse Foundation might soon differ from what has been described here.

//java books /



MODERN JAVA RECIPES

By Ken Kousen

I love recipe books. Judging from the popularity of resources such as StackOverflow, the thirst for working chunks of code that correctly handle a discrete task is nearly insatiable. The big difference between most online sites that offer code solutions and a book such as Ken Kousen's *Modern Java Recipes* is the latter's sustained quality of the content, the detailed explanations, and the ability of one recipe to reference another for developers who don't entirely know what to ask for. The other benefit of a recipe book is that the contents are laid out sequentially, so that variations on a theme are grouped together and you can compare different recipes to obtain a deeper understanding of a problem.

The success of a recipe book rests on the knowledge and diligence of the author. In Kousen, a Java Champion, you have a very knowledgeable developer who is equally expert at presenting the information. There are 74 recipes in 300 pages, which shows the level of detail and background that accompanies every recipe.

A common limitation of recipe books is that you don't remember them when you have a problem they cover or, conversely, you consult them but they don't address the question you have. The better books carefully identify the scope of their contents. This is done in this volume in the subtitle, "Simple Solutions to Difficult Problems in Java 8 and 9"—in other words, recipes for the added features in these two releases of Java. And in fact, this is what you have: the basics (lambdas, method references, functional interfaces); the java.util .function package (consumers, suppliers, predicates, and functions); streams; comparators and collectors; optionals; the java.time package; parallelism and concurrency; and a chapter on Java 9's additions.

A useful addendum is an 18-page appendix that focuses on understanding the complexities of generics in recent releases of Java. Surely, they are now as complex as function declarations in C and C++. Consider **Listing 1** from Kousen's book, which itself is taken from the Java 8 documentation.

This appendix, which saves Kousen from explaining generics syntax repeatedly, is a remarkably clear and thoughtful presentation of a topic that doesn't get enough attention. For expert developers, it will serve as a good refresher.

I like *Modern Java Recipes* a lot and can find little to fault. Any developer working through the subtleties of the features added in Java 8 and Java 9 will find this book a great help. —*Andrew Binstock*

Listing 1.

```
static <T, U extends Comparable <? super U>> Comparator<T> comparing(Function<? Super T,
    ? extends U> keyExtractor)
```

Your Destination for Oracle and Java Expertise

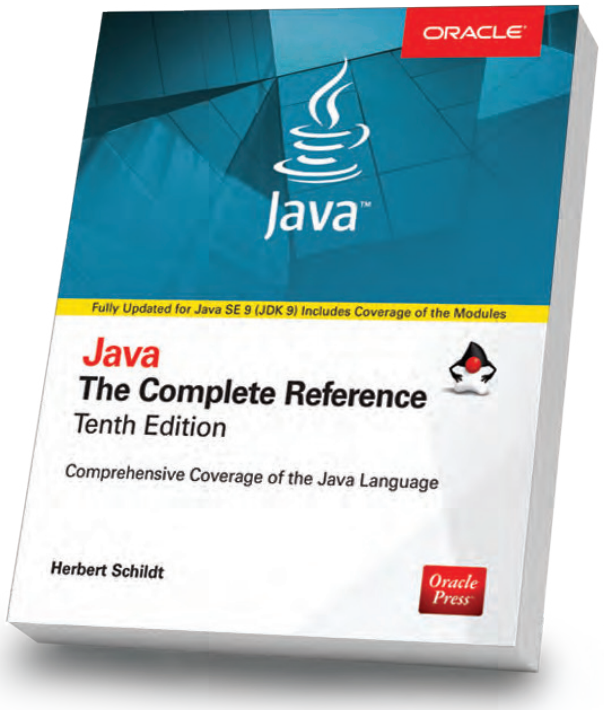
Written by leading experts in Java, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.

ORACLE
Java
Fully Updated for Java SE 9 (JDK 9)
Java
A Beginner's Guide
Seventh Edition
Create, Compile, and Run Java Programs Today
Herbert Schildt
Oracle Press

Java: A Beginner's Guide, 7th Edition

Herb Schildt

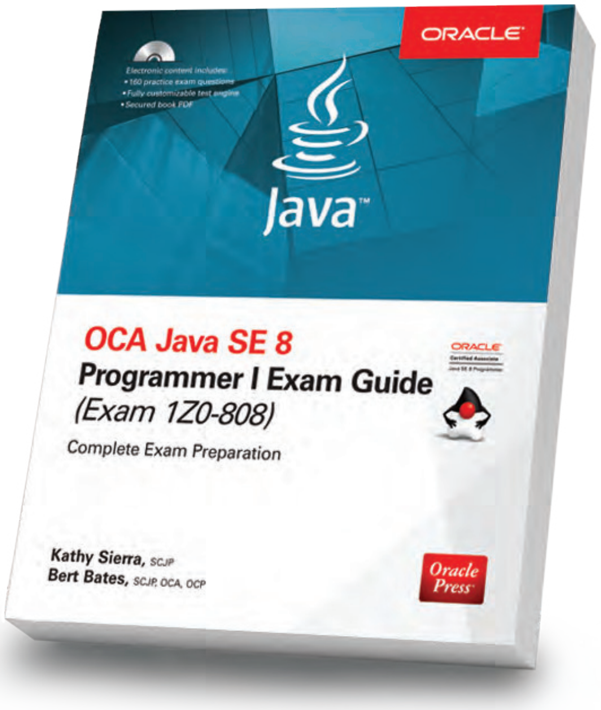
Revised to cover Java SE 9, this book gets you started programming in Java right away.


ORACLE
Java
Fully Updated for Java SE 9 (JDK 9) Includes Coverage of the Modules
Java
The Complete Reference
Tenth Edition
Comprehensive Coverage of the Java Language
Herbert Schildt
Oracle Press

Java: The Complete Reference, 10th Edition

Herb Schildt

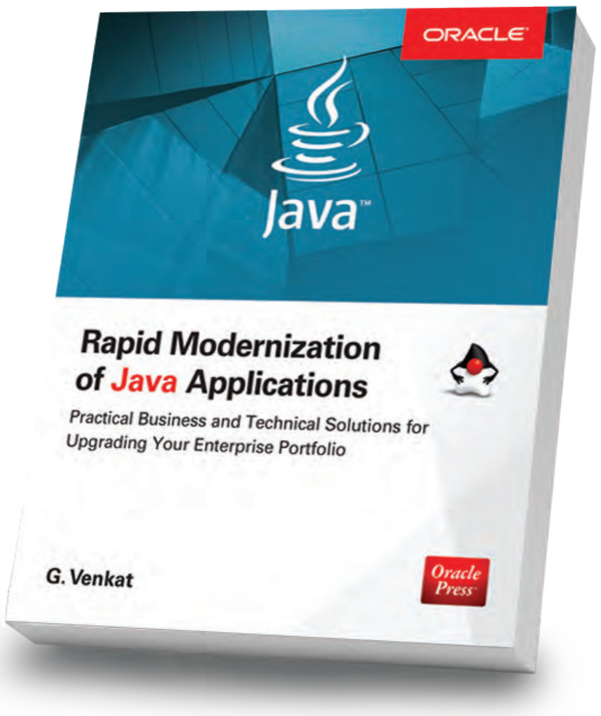
Updated for Java SE 9, this book shows how to develop, compile, debug, and run Java programs.



OCA Java SE 8 Programmer I Exam Guide (Exam 1Z0-808)

Kathy Sierra, Bert Bates

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.


ORACLE
Java
Rapid Modernization of Java Applications
Practical Business and Technical Solutions for Upgrading Your Enterprise Portfolio
G. Venkat
Oracle Press

Rapid Modernization of Java Applications

G. Venkat

Adopt a high-performance enterprise Java application modernization strategy.

Available in print and eBook formats.

www.OraclePressBooks.com • @OraclePress

Jfokus

FEBRUARY 5–7, 2018
STOCKHOLM, SWEDEN
The annual Scandinavian Java developer conference encompasses Java SE and Java EE, front-end and web development, mobile, cloud, IoT, and JVM languages such as Scala and Clojure.

On February 5, the conference will hold the Jfokus VM Tech Summit, which is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects. The schedule for the Summit will be divided equally between traditional presentations of 45 minutes and workshops.

PHOTOGRAPH BY THOMAS FABIAN/FLICKR

Javaneiros

NOVEMBER 18
CAMPO GRANDE, BRAZIL
The Javaneiros conference is a community-organized day of presentations related to Java programming. (No English page available.)

Voxxed Days Thessaloniki

NOVEMBER 23, WORKSHOPS
NOVEMBER 24–25 CONFERENCE
THESSALONIKI, GREECE
Voxxed Days is a community conference for developers, by developers. Sessions include "Spring Boot and Kotlin: A Match Made in Heaven," "Stream Puzzlers: Traps and Pitfalls in Using Java 8 Streams," and "Hands-on Microservices with OpenShift." The two-day conference is preceded by a day of free software-development workshops.

JVM-Con

NOVEMBER 28–29
COLOGNE, GERMANY
Among the topics slated for this conference devoted to JVM languages are Java 9, Java EE, Scala, Jython, Kotlin, Clojure, FreePascal, Groovy, JRuby, cloud native development, microservices, and security. (No English page available.)

DevTernity

DECEMBER 1, KEYNOTES
DECEMBER 2, WORKSHOPS
RIGA, LATVIA
This conference focuses on modern approaches to coding, architecture, testing, engineering scalability, and operations, with 20 speakers presenting in three parallel tracks. The conference is followed by a full day of workshops.

ConFoo

DECEMBER 4–6
VANCOUVER, BRITISH COLUMBIA, CANADA
This multitechnology conference for web developers features targeted sessions for Java and JVM developers. Sessions include presentations on concurrency models with Java 9, Angular/TypeScript, and Practical Symfony 4.

ArchConf

DECEMBER 11, WORKSHOPS
DECEMBER 12–14, CONFERENCE
CLEARWATER, FLORIDA
ArchConf is an educational event for software architects, technical leaders, and senior developers. Topics include agile design, microservices, web application security, cloud architectures, and modular Java.

PHOTOGRAPH BY DOUGLAS SACHA/GETTY IMAGES

CodeMash

JANUARY 9–12, 2018
SANDUSKY, OHIO
CodeMash is a unique event that educates developers on current practices, methodologies, and technology trends in a variety of platforms and development languages such as Java, .NET, Ruby, Python, and PHP.

jSpirit

JANUARY 12–16
HAUSHAM, GERMANY
This is an "unconference"-style event organized by JUG Oberland featuring two days of sessions followed by two days of skiing. Day 3 also has a mini-conference for kids, jSpirit4Kids. Specific topics other than programming in Java are not known in advance.

SnowCamp

JANUARY 24: WORKSHOPS
JANUARY 25–26: CONFERENCE
JANUARY 27: UNCONFERENCE
GRENOBLE, FRANCE
SnowCamp is a developer conference held in the French Alps that focuses on Java, web, cloud, DevOps, and software architecture, with a mix of sessions in French and English. The last day, dubbed "unconference," offers a unique opportunity to socialize with peers and speakers on the ski slopes.

DevConf.cz

JANUARY 26–28
BRNO, CZECH REPUBLIC
DevConf.cz is a free three-day open source developer and DevOps conference. All talks, presentations, and workshops will be conducted in English. Several tracks are devoted specifically to Java EE, and the conference can be attended online.

DeveloperWeek

FEBRUARY 3–4: HACKATHON
FEBRUARY 5: WORKSHOPS
FEBRUARY 5–7: CONFERENCE
FEBRUARY 6–7: EXPO
OAKLAND, CALIFORNIA
DeveloperWeek is the world's largest developer expo and conference series, gathering 8,000 participants for a week-long technology-neutral programming conference and associated events. The theme for 2018 is "Industrial Revolution of Code," and tracks include artificial intelligence, serverless development, blockchain, APIs and microservices, and JavaScript.

Devnexus

FEBRUARY 21–23
ATLANTA, GEORGIA
Devnexus is an international open source developer conference. Its stated goal is to connect developers from all over the world, provide affordable education, and promote open source values. Past presenters have included Venkat Subramaniam, author of Pragmatic's *Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions*.

QCon London

MARCH 5–7: CONFERENCE
MARCH 8–9: WORKSHOPS
LONDON, ENGLAND
Although the content has not yet been announced, past QCon conferences have offered several Java tracks along with tracks related to web development, DevOps, cloud computing, and more. Last year's session topics included performance and low-latency Java.

Voxxed Days Zürich

MARCH 8
ZÜRICH, SWITZERLAND
Voxxed Days Zürich shares the

Devoxx philosophy that content comes first and draws internationally renowned and local speakers. While the schedule of speakers has not yet been announced, last year's event featured presentations on using Java 8 lambdas and StampedLock to manage thread safety, extracting knowledge from cryptocurrencies, and a preview of Java EE 8.

Voxxed Days Melbourne

MAY 2–3
MELBOURNE, AUSTRALIA
Voxxed Days is heading down under to Melbourne, Australia. The event will feature insights into cloud, containers and infrastructure, real-world architectures, data and machine learning, the modern web, and programming languages.

JavaOne 2018

OCTOBER 28–NOVEMBER 1
SAN FRANCISCO, CALIFORNIA
Whether you are a seasoned coder or a new Java programmer, JavaOne is for you. It's the ultimate source of technical information and learning about Java. For five days, Java developers gather from around the world to talk about all aspects of Java and JVM languages, development tools, and trends in programming. Tutorials on numerous related Java and JVM topics will be offered.

JavaOne 2017 Recap

The JavaOne conference in October was headlined by the release of Java SE 9 and the news that Oracle was migrating Java EE to the Eclipse Foundation. These links provide access to some of the popular sessions from the conference:

- "JDK 9 Language, Tooling, and Library Features"
- "Migrating to Modules"
- "JDK 9 Hidden Gems"
- "JUnit 5: New Opportunities for Testing on the JVM"
- "Streams in JDK 8: The Good, The Bad, and The Ugly"
- "Exploring Java 9 with REPL"

Are you hosting an upcoming Java conference that you would like to see included in this calendar? Please send us a link and a description of your event at least 90 days in advance at javamag_us@oracle.com. Other ways to reach us appear on the last page of this issue.

JEP 295: Ahead-of-Time Compilation to Native Binaries

As most Java developers know, Java code is executed by the JVM using one of two primary mechanisms: initially interpretation, and then for code that is used extensively, just-in-time (JIT) compilation. Java does not have ahead-of-time (AOT) compilation in which code is compiled to native executable binaries a priori as is done by C or C++.

JDK Enhancement Proposal (JEP) 295 proposes that Java provide an AOT option. Its primary goal is to reduce the amount of time Java applications spend starting up. Specifically, during the initial parts of a Java program's execution, the JVM has not yet determined what methods to run through the JIT compiler, so the performance does not immediately get the benefit of this step. With the proposed AOT compilation, Java programs would execute binary code from the start. Note that the goal is specifically to reduce that initial delay, rather than achieving higher performance throughout the program's entire run.

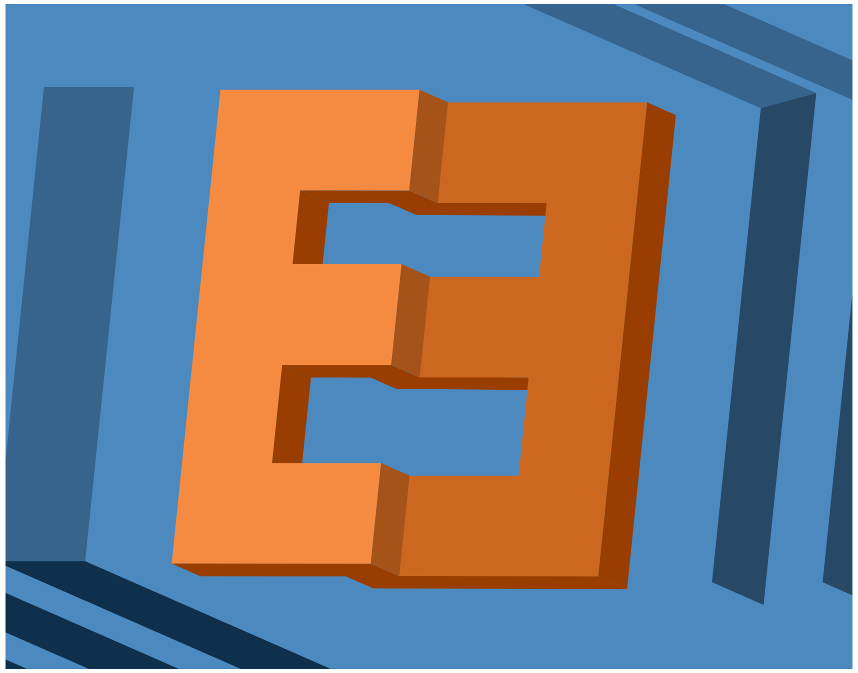
The compiler referenced in this JEP uses Graal as the code-generating back end. At present, the AOT option is being targeted exclusively at 64-bit Linux.

Third-party AOT compilers exist for Java—most notably, Excelsior JET, which delivers binaries for Windows, Linux, macOS, and ARM. Excelsior notes that while the AOT compilation improves startup times, one of the key reasons customers purchase its product is IP protection, as native binaries are more difficult to reverse-engineer than Java bytecode.

Java EE 8: What You Need to Know

While the Java SE community has been focused on the release of Java 9, the Java EE community now has its turn in the spotlight. The editorial at the front of this issue (page 5) examines Oracle's recent announcement that Java EE development is being moved to the Eclipse Foundation.

The articles in this section focus on the many technical advances in Java EE 8. For some technologies, the new release brings significant upgrades and welcome enhancements. These include Servlet 4.0's embrace of HTTP/2 and its new server push capabilities (page 13); CDI 2.0's improved dependency injection (page 23); and JPA 2.2's streaming results, upgraded date conversions, and new annotations (page 43).



ART BY WES ROWELL

We also examine MicroProfile, the new lightweight implementation of Java EE intended for microservices and distributed computing (page 56).

If a single lightweight vehicle isn't enough for you, we look at Java Card, a *super*-lightweight Java SE implementation that thrives on smartcards (page 77). It's interesting to find out how the JVM is activated, how objects' lifetimes are managed, and of course how security is enforced. None of this is easy or trivial in tiny environments.

In addition, we have the final installment of Ben Evans' two-part series on how the JVM executes dynamic method invocations (page 67). Throw in our book review (page 7) and the usual quiz (page 91) with its deep look into the operations of the language, and you have an issue of *Java Magazine* that tops 100 pages. Enjoy! We'll have more coming after this!

ALEX THEEDOM

Servlet 4.0: Doing More Faster

A major new release of the Servlet API embraces the HTTP/2 protocol and anticipates resource requirements.

The long-awaited update to Java EE 8 includes updates to existing APIs—JAX-RS 2.1, Bean Validation 2.0, JavaServer Faces (JSF) 2.3, Contexts and Dependency Injection (CDI) 2.0, JSON with Padding (JSONP) 1.1, and Servlet 4.0—and two brand-new APIs: JSON-Binding (JSON-B) and Java EE Security. Among these APIs, Servlet 4.0 represents a major update, its first since 2009.

The impetus that prompted this major release (rather than a point release) is the worldwide rollout of the HTTP/2 protocol and the many new capabilities it brings. This update to HTTP is the first in nearly 20 years and addresses many of the shortcomings of HTTP 1.x. The new capabilities are numerous (request/response multiplexing, header compression, stream prioritization, and server push), but the most visible feature to users of the Servlet API is Server Push, which I will discuss in this article.

Server Push is not the only noteworthy addition to Servlet 4.0. This release also introduces a feature refinement in the form of the Servlet Mapping API, which permits the runtime discovery of mappings by refining the way referrer paths are obtained. This article discusses these features and how Server Push has been integrated into the JavaServer Faces 2.3 API.

Server Push

Designed to anticipate the resource requirements of a web page, the Server Push feature pushes images, CSS, and JavaScript files, and other resources to the client before request processing has been completed. Therefore, by the time the browser receives the response to its request for the web page, the resources it needs are already in its cache and ready to use.

Gone is the need to shard resources across domains to get around browser TCP connection limitations. Simply specify the resources and let the server handle delivery. This is the way it should always have been.

With the resources already in the browser's cache, the browser can render the page much more quickly. With a typical web page requiring more than 100 resources, it's easy to see how much of an opportunity this represents for performance enhancements.

Server Push in Action

Servlets are the backbone technology behind the Java EE web application tier. They provide the server capabilities that form the basis of many frameworks. JavaServer Faces (JSF) relies on the FacesServlet to manage the request processing lifecycle for web applications, and JSP pages are translated into servlets upon the first client request; so it should be no wonder that servlets are the natural place to expose the HTTP/2 Server Push abstraction.

That abstraction is represented as a `PushBuilder` object and is created by calling the `newPushBuilder()` method from the `HttpServletRequest` instance passed to all overridden request handling methods.

With a `PushBuilder` instance, you can start pushing the resources required by the requested web page. The resource is set on the `PushBuilder` instance by passing its location to the `path()` method. The resource is pushed to the client by invoking the `push()` method. It can be reused to send as many resources as required.

Listing 1 and Listing 2 show the simplest example that uses Server Push. Listing 1 shows an HTTP servlet that responds to a GET request to the URI `/simplestexample`:

Listing 1.

```
@WebServlet("/simplestexample")
public class SimplestExample extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest request,
          HttpServletResponse response)
          throws ServletException, IOException {
```

```
    request.newPushBuilder()
          .path("images/coffee-cup.jpg")
          .push();
    getServletContext()
          .getRequestDispatcher("/coffee-cup.jsp")
          .forward(request, response);  }}
```

Listing 2 shows the JSP web page:

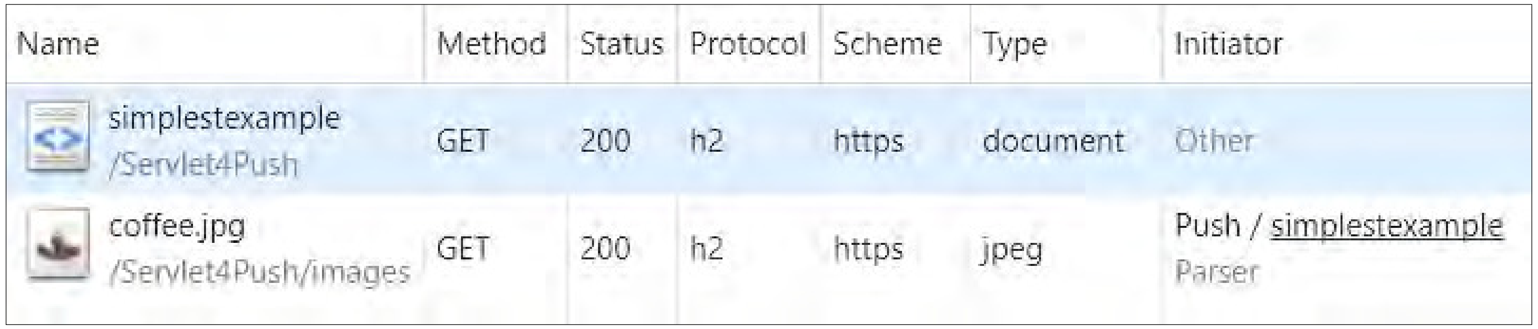
Listing 2.

```
<html>
<head>
    <title>Coffee-Cup</title>
</head>
<body>
    <img src='images/coffee-cup.jpg'>
</body>
</html>
```

In Listing 1, I have a servlet called `SimplestExample` and a JSP to which the servlet dispatches. As you can see, the JSP page requires only one resource, the `coffee-cup.jpg` image. When the `doGet()` request handling method is invoked, it creates a new `PushBuilder` instance, sets the image's location, and calls the `push()` method to send the resource to the client.

As the image is making its way to the client, the servlet is forwarding the request to the JSP that requires the `coffee-cup.jpg` resource. By the time the browser has received the rendered HTML, it already has the image in its cache and can display the page without needing to make another request.

To see Server Push in action, you can use the developer's tools provided by Google's Chrome browser. Select More Tools>Developer Tools and then click the Network tab, and make a request to the `SimplestExample` servlet. You should see a result similar to that shown in Figure 1. You can



Name	Method	Status	Protocol	Scheme	Type	Initiator
simplestexample /Servlet4Push	GET	200	h2	https	document	Other
coffee.jpg /Servlet4Push/images	GET	200	h2	https	jpeg	Push / simplestexample Parser

Figure 1: Resource request satisfied by Server Push

clearly see that the protocol it uses is h2 (short for "HTTP/2") and the image was initiated via Push. This confirms that Server Push was used to satisfy the resource request.

TLS Required for HTTP/2

You might have noticed in **Figure 1** that the scheme of the request is HTTPS. This is because all major browser vendors have chosen to implement HTTP/2 over Transport Layer Security (TLS) only. However, the specification does not mandate that a secure connection is required for successful HTTP/2 communication. Browser vendors have made that decision on our behalf.

Care must be taken when using a new `PushBuilder` object. A call to `newPushBuilder()` will return `null` if the connection is not secure, if the client does not support Server Push, or if the client has requested to disable Server Push via the SETTINGS_ENABLE_PUSH parameter of a SETTINGS frame.

If you want to try this example for yourself, you can clone the code from the GitHub repository.

Anatomy of a PushBuilder

Each new instance of `PushBuilder` created by calling `newPushBuilder()` is based upon the current `HttpServletRequest` instance. It is initiated with the HTTP GET method, and all headers are stripped out except for `conditional`, `range`, `expect`, `authorization`, and `referrer` headers.

`PushBuilder` implements the builder pattern where chained method calls are used to mutate the instance before calling the `push()` method. The resource path is the only configuration required before pushing. The push action initiates an asynchronous nonblocking request, and when it returns, the path and conditional headers are cleared in preparation for the builder's reuse.

You might be wondering how useful it is to use PushBuilder in this way in a real-world application, and you would be right to question this. The example was a little artificial and is for demonstration purposes only. A more likely scenario is that your application will use JSF or JSP. So let's look at JSF integration and how you might use the Server Push feature with JSP.

Especially welcome is Server Push's seamless integration into the JSF API, which makes the adoption of performance-enhancing HTTP/2 features possible without any code changes.

JSF Integration

The JSF API takes full advantage of Server Push. It has full knowledge of all the resource requirements of the requested web page and is well placed to push resources to the client. Listing 3 shows a simple example of a JSF page that relies on three resources:

Listing 3.

```
<h:outputStylesheet library="css"
                    name="coffee-cup.css"/>
<h:outputScript library="js" name="coffee-cup.js"
                target="head"/>
<h:head>
     <title>JSF 2.3 Server Push Example</title>
</h:head>
<h:body>
     <h:form>
          <h:graphicImage library="images" name="coffee-cup.jpg"/>
     </h:form>
```

```
</h:body>
</html>
```

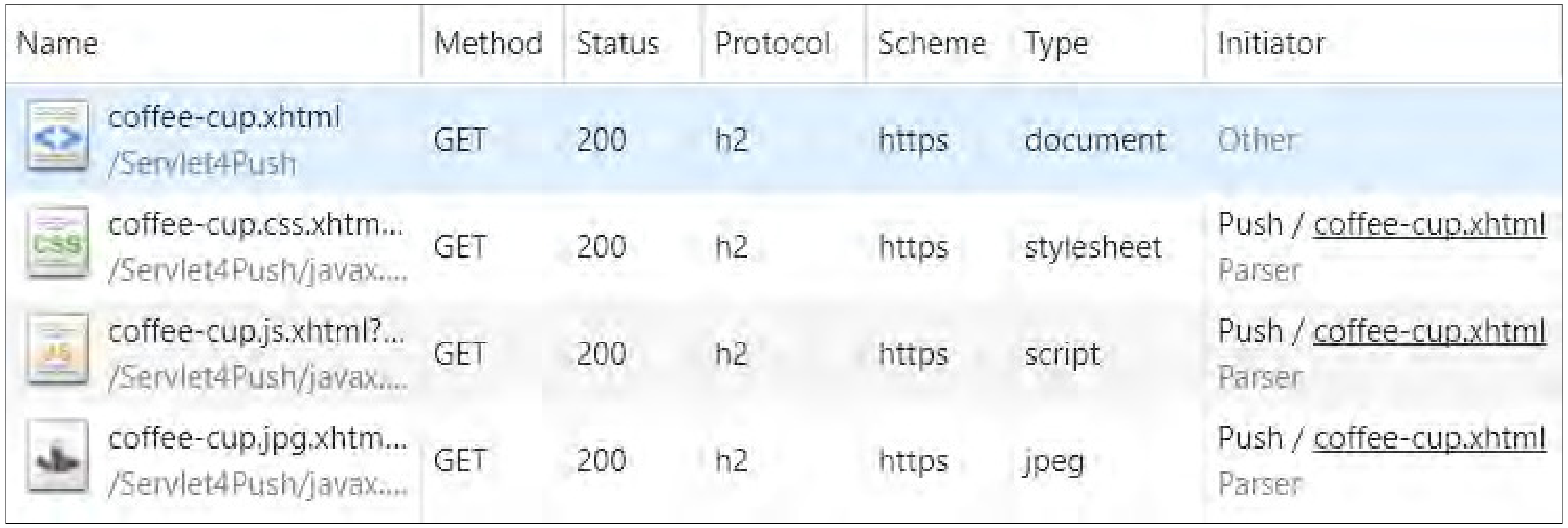
As you can see, the JSF web page requires an image, a CSS file, and a JavaScript file. When the page is requested, all three resources will be pushed to the client via Server Push before the rendered page is returned. This can be clearly seen in Chrome's Network tab, as shown in Figure 2.

The page's resources are pushed during the JSF RenderResponsePhase lifecycle phase. For each resource that is identified during this process, the `ExternalContextImpl.encodeResourceURL()` method is invoked and passed the resource location. A new `PushBuilder` object is created from the `HttpServletRequest` instance associated with the `ExternalContext`, and the resource is pushed to the client. All inline resources are pushed to the client, and all such pushes are initiated before the page is rendered to the client.

If you want to try this, the code can be cloned from the GitHub repository.

JSP Integration

So what about JSP? Disappointingly, JSP has not been updated to benefit from Server Push. This is unfortunate considering the number of active web applications still using JSP. However,



Name	Method	Status	Protocol	Scheme	Type	Initiator
coffee-cup.xhtml /Servlet4Push	GET	200	h2	https	document	Other
coffee-cup.css.xhtm... /Servlet4Push/javax....	GET	200	h2	https	stylesheet	Push / coffee-cup.xhtml Parser
coffee-cup.js.xhtml?... /Servlet4Push/javax....	GET	200	h2	https	script	Push / coffee-cup.xhtml Parser
coffee-cup.jpg.xhtm... /Servlet4Push/javax....	GET	200	h2	https	jpeg	Push / coffee-cup.xhtml Parser

Figure 2. JSF inline resource pushed to client

this does not mean that your JSP web applications are doomed to HTTP 1.1 slowness; there is a solution, albeit a bespoke solution.

You know that JSP pages are translated into servlets and requests to those servlets can be intercepted with web filters. With this knowledge, you can create a solution that allows you to harness the power of Server Push.

The runtime discovery of the URL mapping that causes a servlet to be activated has been refined, thanks to the new Servlet Mapping API.

By implementing a web filter, all requests to the web application can be intercepted and the intercepting filter maintains a cache of resource locations for each page requested. On subsequent requests, the resources the page requires are pulled from the cache and pushed to the client before the filter forwards to the JSP translated servlet (or to the next filter in the chain).

The way this works is that the first time a page is requested, the resources it needs are identified. It is assumed that soon after the initial page request is made, the browser will start asking for the resources it needs. These resource requests are identified by examining the referrer header and matching the referrer page name to the name of the initial page request, thus building up a cache of page-to-resource locations.

This is an effective solution that brings the Server Push feature to any part of your application that does not support it out of the box. In fact, this is the solution that Jetty version 9 has implemented in its PushCacheFilter web filter.

The `HttpServletRequest` is the gateway to the Server Push filter, which means that anywhere you find the `HttpServletRequest` instance you can use a new instance of `PushBuilder` and start pushing resources.

Runtime Discovery of Mappings

The runtime discovery of the URL mapping that causes a servlet to be activated has been refined, thanks to the new Servlet Mapping API. Frameworks that need to know the exact mapping that caused the servlet activation are the most likely to benefit from this feature. For example, a request to file.ext, /path, and /path/to/file.ext activates the servlet with URL patterns /path/* and *.ext.

A reference to the Servlet Mapping API is obtained from the servlet's HttpServletRequest instance by calling the getHttpServletMapping() method. The API exposes four methods:

- The getMatchValue() method that returns the value that was matched
- The getPattern() method that returns the URL pattern that matched the request
- The getMappingMatch() method that returns the type of the match and is represented as an enum with one of the following values: CONTEXT_ROOT, DEFAULT, EXACT, EXTENSION, IMPLICIT, PATH, and UNKNOWN
- The getServletName() method that returns the fully qualified name of the servlet that was activated with the request

Listing 4 shows runtime discovery of mappings:

Listing 4.

```
@WebServlet({"/path/*", "*.ext"})
public class ServletMapping extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                   HttpServlet Response response)
                   throws ServletException,
                   IOException {

    HttpServletMapping servletMapping =
        request.getHttpServletMapping();

    response.getWriter()
          .append("<html><body>")
          .append("Value Matched: ")
          .append(servletMapping.getMatchValue())
          .append("<br/>")
          .append("Pattern Used: ")
          .append(servletMapping.getPattern())
          .append("<br/>")
          .append("Mapping Matched: ")
          .append(servletMapping.getMappingMatch()
               .name())
          .append("<br/>")
          .append("Servlet Name:")
          .append(servletMapping.getServletName())
          .append("<br/>")
          .append("</body></html>");
    }
}
```

The output of the code in **Listing 4** is shown in **Table 1**.

The servlet name returned by this example is `com.readlearncode.servlet4.mapping.ServletMapping`.

Other Notable Updates in Servlet 4.0

Arguably, Server Push and the Servlet Mapping API are the most significant additions to the Servlet 4.0 release; nevertheless, I would be remiss if I did not mention other changes and additions.

Support has been added for HTTP Trailer, and a new `GenericFilter` and `HttpFilter` have also been added. These additions simplify writing filters. The `GenericFilter` provides minimal implementations of the lifecycle methods `init` and `destroy`.

	file.ext	**/path**	**/path/to/file.ext**
MAPPING MATCHED	file	path	to/file.ext
VALUE MATCHED	*.ext	/path/*	/path/*
PATTERN USED	EXTENSION	PATH	PATH

Table 1. Output from Listing 4

In addition, ServletContext gets a few new methods. The addJspFile() method adds the servlet with the given JSP file to the servlet context. The session timeout and the default request character encoding for the current servlet context get mutator and accessor methods.

Java SE 8 is now the minimum version with which servlet containers must be built. With this change, there has been an upgrade to Java 8 features such as the addition of default methods to Listener interfaces.

The HttpServletRequestWrapper.isRequestedSessionIdFromUrl() method has been deprecated, and there has been some modification and clarification to the corresponding Javadoc with regard to various methods and XML configurations.

Spring Framework 5.0

Spring Framework version 5.0, which has just been released, boasts HTTP/2 support natively with Tomcat, Jetty, and Undertow. Full support for Servlet 4.0 is expected in Spring Framework 5.1. Nevertheless, the framework fully supports Server Push's capabilities.

Conclusion

Two headline features, Server Push and the Servlet Mapping API, are welcome additions to the Servlet API. Especially welcome is Server Push's seamless integration into the JSF API, which makes the adoption of performance-enhancing HTTP/2 features possible without any code changes. </article>

Alex Theedom (@readlearncode) is an instructor at LinkedIn Learning. He is the author of *Java EE 8: Only What's New* (Leanpub.com) and coauthor of *Professional Java EE Design Patterns* (Wrox Press, 2015). He blogs profusely at readlearncode.com about Java EE. When he's not in front of the computer screen, you can often find him presenting at conferences on Java EE–related topics.

ARJAN TIJMS

CDI 2.0: Making Dependency Injection a Lot Easier

A new spec, new features, and new annotations—what's not to like?

Contexts and Dependency Injection (CDI) is Java EE's primary dependency injection framework. It was introduced with Java EE 6 in 2009. Started by Gavin King (of Hibernate fame) and originally intended to unite the JavaServer Faces (JSF) and Enterprise JavaBeans (EJB) bean models, CDI is now slowly but steadily becoming the backbone of all of Java EE. JSF 2.3, for instance, has fully deprecated its own managed bean model and dependency injection in favor of CDI, while the new Java EE Security API has been designed specifically to work with CDI. Interceptors and bean validation are usable without CDI, but they are both easier to use with CDI.

Java EE 8 delivered a major update with CDI 2.0. Central to this update was splitting the spec into three parts: the core spec, Java SE features, and Java EE features. This division was done primarily to standardize how to use CDI in Java SE, but it can also be seen as an attempt to make CDI a more-core, fundamental technology. For instance, the built-in beans for Java EE—which make, for example, the injection of `HttpServletRequest` or `Principal` possible—could be moved to a more appropriate spec in Java EE without affecting the core part of the spec or the Java SE part of CDI.

In this article, I demonstrate some of the most useful features in CDI 2.0, including

- Simplified programmatic creation of beans
- Programmatically adding an interceptor (to a built-in CDI bean)
- Programmatic bean lookup simplifications

There are some other interesting new features in CDI 2.0, such as event ordering and asynchronous events, which will be covered in future articles in this magazine. As you can tell if you've gotten this far, this article is aimed at developers already familiar with Java EE and CDI.

Simplified Programmatic Creation of Beans

CDI has the well-known concept of *producers*. Simply put, a producer is a kind of general factory method for some type. It's defined by annotating a method with `@Produces`. An alternative "factory" for a type is simply a class itself; a class is inherently a factory of objects of its own type. In CDI, both of these kinds of factories are represented by the `Bean<T>` type. The name might be somewhat confusing, but a `Bean<T>` in CDI is, thus, not a bean itself but a type used to create instances (that is, a factory). An interesting aspect of CDI is that those `Bean<T>` instances are not just internally created by CDI after encountering class definitions and producer methods; they can be added programmatically by user code as well.

Via this mechanism, it's possible to dynamically register these "factories." This ability can be handy in a variety of cases, for instance, when many similar producer methods would have to be defined statically or when generic producers are needed. As it stands, generics are not particularly well supported in CDI. Instead of trying to create a somewhat generic producer, an alternative strategy could be to actually scan which types an application is using and then dynamically create a `Bean<T>` for each type.

In CDI 2.0, you can take advantage of the new InterceptionFactory to bind a library-shipped interceptor to a library-shipped built-in bean.

Programmatically adding a `Bean<T>` instance and using CDI producers are techniques that overlap somewhat in functionality. The difference is that CDI producers essentially make only the "create instance" aspect dynamic; the rest (such as scope, types, and so on) is more or less static. However, a programmatically added `Bean<T>` makes all those aspects dynamic. The downside of a `Bean<T>` is that it has to be added via a CDI extension, which runs only when CDI starts up.

In CDI 1.x, programmatically adding `Bean<T>` is quite a bit of work and a bit complex, because it requires you to decide what to return as a default for various methods that are not directly of interest.

CDI 2.0 has addressed that by providing a very convenient builder that not only makes creating a Bean<T> instance far less verbose but also takes away most of the guesswork. Consider the simple interface shown in the following code:

```
public interface MyBean {
    String sayHi();
}
```

And a class implementing that interface:

```
public class MyBeanImpl implements MyBean {

    private final String greet;

    public MyBeanImpl(String greet) {
        this.greet = greet;
    }

    @Override
    public String sayHi() {
        return greet;
    }
}
```

Note that this class has no default constructor, so CDI normally would not be able to use this class directly as a bean.

Now consider the following CDI extension:

```
public class CdiExtension implements Extension {

    public void afterBean(
      @Observes AfterBeanDiscovery afterBeanDiscovery) {
        afterBeanDiscovery
            .addBean()
            .scope(ApplicationScoped.class)
            .types(MyBean.class)
            .id("Created by " + CdiExtension.class)
            .createWith(e -> new MyBeanImpl("Hi!"));
    }
}
```

This code makes an @ApplicationScoped bean available for injection into MyBean injection points, backed by the MyBeanImpl class. In the example shown above, the MyBeanImpl type is completely hidden and CDI will inject only into MyBean and nothing else. If there were a need to inject into other (related) types, the other types can be provided in the types() method of the builder as well.

As with all CDI extensions, the extension class has to be registered by putting its fully qualified name (FQN) in META-INF/services/javax.enterprise.inject.spi.Extension.

Once all this is done, the bean can be injected just like any other bean:

```
@ApplicationScoped
public class SomeBean {

    @Inject
    private MyBean myBean;

}
```

Programmatically Adding an Interceptor to a Built-In CDI Bean

In CDI, beans can be augmented via two artifacts: decorators and interceptors.

- Decorators are typically owned by the application code and can decorate a bean that's shipped by the container (built-in beans) or a library.

- Interceptors are typically shipped by a library and can be applied (bound) to a bean that's owned by the application.

So, how do you bind a library-shipped interceptor to a library-shipped built-in bean? In CDI 1.2 and before, this wasn't really possible, but in CDI 2.0, you can take advantage of the new InterceptionFactory to do this. It's not entirely trivial yet, but it's doable. Here is how to apply the @RememberMe interceptor binding from the new Java EE 8 Security specification to a built-in bean of type HttpAuthenticationMechanism, which is from the Java EE Security spec as well.

First, configure the authentication mechanism by means of the following annotation:

```
@BasicAuthenticationMechanismDefinition(
    realmName="foo"
)
```

This annotation causes the container to enable a built-in bean with an interface type of HttpAuthenticationMechanism, but having an unknown (vendor-specific) implementation. The annotation can be placed on almost any class on the classpath.

Next, define an alternative for this bean via a CDI producer:

```
@Alternative
@Priority(500)
@ApplicationScoped
public class ApplicationInit {

    @Produces
    public HttpAuthenticationMechanism produce(
        InterceptionFactory<HttpAuthenticationMechanismWrapper>
            interceptionFactory,
        BeanManager beanManager) {
            return ...
    }
}
```

Note, perhaps somewhat counterintuitively, that the @Alternative annotation is put on the bean hosting the producer method, not on the producer method itself. A small challenge here is to obtain the bean with type HttpAuthenticationMechanism that would have been chosen by the CDI runtime had the producer not been there. For a decorator, this is easy because CDI makes that exact bean injectable via the @Decorated qualifier. Here, this will have to be done manually. One way is to get all the beans of type HttpAuthenticationMechanism from the bean manager (this will include both alternatives and nonalternatives), filter the producer bean (ApplicationInit) from that set, and then let the bean manager resolve the set to the one that would be chosen for injection. After that, a reference is created for that chosen bean.

The following shows this process in code:

```
HttpAuthenticationMechanism mechanism =
    createRef(
        beanManager.resolve(
            beanManager
                .getBeans(HttpAuthenticationMechanism.class)
                .stream()
                .filter(e -> !e.getBeanClass()
                     .equals(ApplicationInit.class))
                .collect(toSet())), beanManager);
```

Note that the code filters on the getBeanClass() outcome. The BeanClass, perhaps somewhat confusingly, does not necessarily represent the class (or classes) of the bean itself but rather where the bean is created. For a producer, this is the bean that contains the producer method, so that is a very good handle for filtering out the "current" producer bean.

The createRef() method is defined as follows:

```
HttpAuthenticationMechanism createRef(
    Bean<?> bean, BeanManager beanManager) {
        return (HttpAuthenticationMechanism) beanManager.getReference(
            bean,
            HttpAuthenticationMechanism.class,
            beanManager.createCreationalContext(bean));
}
```

You now have an instance of the bean on which you can apply the interceptor binding. Unfortunately, there's a somewhat peculiar and very serious note in the CDI spec regarding the method that creates a proxy with the required interceptor attached: "If the provided instance is an internal container construct (such as client proxy), non-portable behavior results."

Because the `HttpAuthenticationMechanism` is a client proxy (it's application-scoped by spec definition), you have no choice but to provide an extra wrapper. The interceptor will be applied to the wrapper, then, and the wrapper will delegate to the actual `HttpAuthenticationMechanism` instance, for example:

```
HttpAuthenticationMechanismWrapper wrapper =
    new HttpAuthenticationMechanismWrapper(mechanism);
```

The wrapper is defined straightforwardly, as follows:

```
public class HttpAuthenticationMechanismWrapper
        implements HttpAuthenticationMechanism {

    private HttpAuthenticationMechanism httpAuthenticationMechanism;

    public HttpAuthenticationMechanismWrapper() {
    }

    public HttpAuthenticationMechanismWrapper(
            HttpAuthenticationMechanism httpAuthenticationMechanism) {
```

```
    this.httpAuthenticationMechanism = httpAuthenticationMechanism;
}

HttpAuthenticationMechanism getWrapped() {
    return httpAuthenticationMechanism;
}

@Override
public AuthenticationStatus validateRequest(
        HttpServletRequest request, HttpServletResponse response,
        HttpMessageContext httpMessageContext)
        throws AuthenticationException {

    return getWrapped().validateRequest(
                            request,
                            response,
                            httpMessageContext);
}

@Override
public AuthenticationStatus secureResponse(
        HttpServletRequest request, HttpServletResponse response,
        HttpMessageContext httpMessageContext)
        throws AuthenticationException {

    return getWrapped().secureResponse(
                            request,
                            response,
                            httpMessageContext);
}
```

```
    @Override
    public void cleanSubject(HttpServletRequest request,
            HttpServletResponse response,
            HttpMessageContext httpMessageContext) {

        getWrapped().cleanSubject(
                        request,
                        response,
                        httpMessageContext);
    }
}
```

Having the `HttpAuthenticationMechanism` instance ready, the annotation instance can now be configured dynamically. The instance has to be created first, which can be done via CDI's provided `AnnotationLiteral` helper type. Because the `@RememberMe` annotation has many attributes, it's a little unwieldy, but it's still relatively straightforward:

```
public class RememberMeAnnotationLiteral extends
    AnnotationLiteral<RememberMe> implements RememberMe {

    private static final long serialVersionUID = 1L;

    int cookieMaxAgeSeconds;
    String cookieMaxAgeSecondsExpression;
    boolean cookieSecureOnly;
    String cookieSecureOnlyExpression;
    boolean cookieHttpOnly;
    String cookieHttpOnlyExpression;
    String cookieName;
    boolean isRememberMe;
    String isRememberMeExpression;

    public RememberMeAnnotationLiteral(
        int cookieMaxAgeSeconds,
        String cookieMaxAgeSecondsExpression,
        boolean cookieSecureOnly,
        String cookieSecureOnlyExpression,
        boolean cookieHttpOnly,
        String cookieHttpOnlyExpression,
        String cookieName,
        boolean isRememberMe,
        String isRememberMeExpression) {

        this.cookieMaxAgeSeconds =           cookieMaxAgeSeconds;
        this.cookieMaxAgeSecondsExpression = cookieMaxAgeSecondsExpression;
        this.cookieSecureOnly =              cookieSecureOnly;
        this.cookieSecureOnlyExpression =    cookieSecureOnlyExpression;
        this.cookieHttpOnly =                cookieHttpOnly;
        this.cookieHttpOnlyExpression =      cookieHttpOnlyExpression;
        this.cookieName =                    cookieName;
        this.isRememberMe =                  isRememberMe;
        this.isRememberMeExpression =        isRememberMeExpression;
    }

// + getters for instance variables


}
```

With that definition in place, the annotation can be added to the interception factory with programmatically configured attribute values:

```
interceptionFactory.configure().add(
    new RememberMeAnnotationLiteral(
        86400, "",          // cookieMaxAgeSeconds
        false, "",          // cookieSecureOnly
        true, "",           // cookieHttpOnly
        "JREMEMBERMEID", // cookieName
        true, ""            // isRememberMe
    )
);
```

Finally, the above-mentioned new proxy can be created with the configured interceptor binding applied to it using the interception factory's createInterceptedInstance() method and return this from the produce() method of the ApplicationInit bean that was shown at the start of this section:

```
return interceptionFactory.createInterceptedInstance(
    new HttpAuthenticationMechanismWrapper(wrapper));
```

Note that there's a small caveat here: if the Interceptor needs access to the interceptor bindings (which is almost always the case when the binding has attributes), you cannot just inspect the target type as you would usually do in CDI 1.2 and earlier code. The interceptor binding annotation is not physically present on the type. At present, it's not entirely clear how to obtain these in a portable way. The interceptors in the Java EE Security reference implementation (Soteria) use a reference-implementation-specific way for now.

As a second example, I'll demonstrate another use case where the Java Transaction API (JTA) @Transactional annotation is dynamically applied to a single method of a bean.

For this example, consider a simple Employee entity defined as follows:

```
@Entity
public class Employee {
    @Id
    @GeneratedValue
    private int id;

    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

And consider a service that persists that entity, which is defined as:

```
@ApplicationScoped
public class EmployeeService {

    @Inject
    private EntityManager entityManager;

    public void persist(Employee employee) {
        entityManager.persist(employee);
    }

    public Employee getById(int id) {
        return entityManager.find(Employee.class, id);
    }
}
```

For completeness, a resource producer field and a `persistence.xml` file are needed, which are defined as follows. The producer field:

```
 @Produces @PersistenceContext
 private EntityManager entityManager;
```

The `persistence.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/
xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="MyPU">
        <properties>
            <property
                name="javax.persistence.schema-generation.database.action"
                value="drop-and-create" />
        </properties>
    </persistence-unit>
```

```
</persistence>
```

The `EmployeeService` shown above is a plain CDI bean, and it doesn't have any transactional semantics of its own. Yet, per the requirements of `EntityManager#persist`, a transaction is required for that method to do its work. Possibly, the service has been written with the assumption that the client starts that transaction. If the service wasn't, you modify the code directly to add `@Transactional` to the `persist()` method, and the CDI 2.0 `InterceptionFactory` can be used again to dynamically add it.

The full producer method is shown below:

```
@Produces
public EmployeeService produce(
        InterceptionFactory<EmployeeService> interceptionFactory,
        BeanManager beanManager) {

    EmployeeService employeeBean = createRef(beanManager.resolve(
            beanManager.getBeans(EmployeeService.class)
                       .stream()
                       .filter(e -> !e.getBeanClass()
                                      .equals(ApplicationInit.class))
                       .collect(toSet())), beanManager);

    interceptionFactory.configure()
                       .filterMethods(
                           am -> am.getJavaMember()
                                   .getName()
                                   .equals("persist"))
                       .forEach(
                           amc -> amc.add(new TransactionLiteral()));
    return interceptionFactory.createInterceptedInstance(
            new EmployeeBeanWrapper(employeeBean));
}
```

Notice that I have now added the `filterMethods()` part to select all methods with the name "persist." For this service, that would be only one method. Subsequently, the code adds the `@Transaction` annotation to that method via a `TransactionLiteral`, which works in the same way as the `RememberMeAnnotationLiteral` shown above. I also need a wrapper again, which this time is created by subclassing the service instead of using an interface. It's shown below:

```
public class EmployeeBeanWrapper extends EmployeeService {

    private EmployeeService employeeBean;

    public EmployeeBeanWrapper() {
    }

    public EmployeeBeanWrapper(EmployeeService employeeBean) {
        this.employeeBean = employeeBean;
    }

    EmployeeService getWrapped() {
        return employeeBean;
    }

    @Override
    public void persist(Employee employee) {
        getWrapped().persist(employee);
    }

    @Override
    public Employee getById(int id) {
        return getWrapped().getById(id);
    }
}
```

Note that in the produce() method shown in the code immediately before this latest code, I obtained an instance of the original EmployeeService class using CDI instead of just instantiating via the new operator. This approach is needed for the injection in that bean to do its work and to apply the correct scope.

Earlier, I mentioned a caveat: it is difficult for an Interceptor implementation to access the interceptor bindings when they have been added dynamically. That same caveat applies here. The current JTA Interceptor implementations are not yet aware of CDI 2.0 and will not see the dynamic annotation, and thus they will not be able to read its attributes. Payara 5 will address this, and I expect other implementations will soon do the same thing.

You should also be aware that the add() methods of the InterceptionFactory will happily accept any type of annotation literal, but only those representing interceptor bindings will have any effect. For instance, interceptionFactory.configure().add(SessionScoped.Literal.INSTANCE) will both compile and not throw any exceptions at runtime, but the produced bean will *not* be given the configured scope.

While CDI is widely known for its ability to inject beans into injection points, it also features an API to look up beans programmatically.

Programmatic Bean Lookup Simplifications

While CDI is widely known for its ability to inject beans into injection points, it also features an API to look up beans programmatically. This API largely mimics the way injection works. That is, if you can create an injection point of type Foo with qualifier Bar, then the same bean that would have been injected there can also be looked up by passing Foo.class and a Bar annotation instance to this API.

As seen previously, though, annotation instances are somewhat tedious to create. So, CDI 2.0 has introduced default annotation instances and builders for most of its annotations. A short example was already shown earlier. Instead of painstakingly creating an annotation instance for @SessionScoped, you can use SessionScoped.Literal.INSTANCE.

For annotations that have attributes, convenient builders have been provided by CDI 2.0, and for the important annotations that CDI does not own (those from JSR 330, which had little hope of being updated during Java EE 8), CDI has provided annotation literal types too, for example, for @Named.

A few other smaller conveniences for lookup have been added as well. For example, when you are doing a lookup, it's now possible to check via one test whether the lookup resolved to an instance, instead of testing for various different failure modes (such as ambiguity, no bean found, and so forth).

Finally, JDK 8 support has been added, such as getting streams for a lookup that can result in many candidates. Let's look at examples of these. Consider the following simple interface:

```
public interface MyGreeting {
    String getGreet();
}
```

And two different implementations of this:

```
@Named("northern")
public class MyGreeting1 implements MyGreeting {
    @Override
    public String getGreet() {
        return "Ay-up!";
    }
}
```

And consider this:

```
@Named("informal")
public class MyGreeting2 implements MyGreeting {
    @Override
    public String getGreet() {
        return "Hiya!";
    }
}
```

With CDI 2.0, you can easily look up all `MyGreeting` candidates and process these via a JDK 8 stream:

```
Instance<MyGreeting> myGreetings =
    CDI.current().select(MyGreeting.class);

myGreetings.stream()
           .forEach(e -> out.println(e.getGreet()));
```

This code prints “Ay-up!” and “Hiya!” to the standard output. Because it’s a normal JDK 8 stream, any other operation can be applied to it if needed, such as filtering, mapping, and so on. Selecting a single specific instance from the `myGreetings` preselection can be done as follows;

```
MyGreeting greeting =
    myGreetings.select(NamedLiteral.of("northern")).get();
```

Or, without the preselection, it can be done in one step:

```
MyGreeting greeting =
    CDI.current()
```

```
        .select(MyGreeting.class, NamedLiteral.of("northern"))
        .get();
```

CDI 2.0 has made working with beans in a programmatic way much easier, and it has enabled things that weren't possible before in CDI.

Both versions of the code will result in the MyGreeting1 instance being resolved. The existence of the new NamedLiteral class makes it relatively easy to do this lookup.

Of course, a bean might not be available for several reasons. For example, a bean of the requested type might exist but not have the requested qualifier. Or, if the requested type is an interface, it might be implemented by two or more classes, which makes it impossible to return a single instance. To test for any kind of failure to obtain a bean, CDI 2.0 introduced the new isResolvable() method. The following example shows how this is used:

```
boolean isResolvable =
    myGreetings.select(NamedLiteral.of("formal"))
               .isResolvable()
```

Because there isn't a MyGreeting implementation available with a @Named("formal") qualifier, the result of the test above will be false.

Conclusion

CDI 2.0 has made working with beans in a programmatic way much easier, and it has enabled things that weren't possible before in CDI, such as combining interceptor bindings that are provided by a container or library with beans that are also provided by a container or library. We've seen that combining these provided interceptor bindings and beans takes a bit of code, but that code should be easy to adapt for your own projects.

The new annotation instances are a great addition, and make requesting beans from the bean manager much easier. You should be aware, though, that in Java EE 8, they are available only for CDI annotations from the CDI spec itself and not for CDI annotations originating from other specs (such as JSF, Java EE Security, JTA, and so on) with the exception of JSR 330 annotations.

Existing interceptors both inside and outside Java EE that require access to an annotation's attributes might need to be updated to take into account that interceptors can be added dynamically and that the interceptor binding annotations are not necessarily actually present on the class or its methods. Up until now, this hasn't really been done much (for example, the reference implementation of the Java EE Security API supports this in a limited way, but the spec makes no mention of it).

All in all, CDI 2.0 is another great step forward and offers a lot of useful new features. </article>

Arjan Tijms works for Payara Services on the next-generation Payara 5 server, and he is a JSF (JSR 372) and Security API (JSR 375) Expert Group member. He is a cocreator of the popular OmniFaces library for JSF that won a 2015 Duke's Choice Award, and he is the main creator of a set of tests for the Java Authentication Service Provider Interface for Containers (JASPIC), which has been used by various Java EE vendors. Tijms holds an MSc in computer science from the University of Leiden in the Netherlands.

THE NLJUG

The Netherlands Java User Group, better known as the NLJUG, has a national reach throughout the Netherlands. One of the largest JUGs in Europe, it currently has more than 4,300 members and 58 business partners.

The NLJUG is best known for its J-Fall conference, the leading event of its kind for the Dutch-speaking Java community. In addition to J-Fall, the NLJUG organizes J-Spring; the IoT Tech Day; and the Masters of Java, a Java "funprogging" contest.

It also publishes its own Java magazine for members six times a year, featuring articles from both the Dutch Java community and international authors.

The JUG participates in the Java Community Process (JCP) through the Adopt-a-JSR program. It was nominated as Outstanding Adopt-a-JSR Participant in the JCP Awards in 2016 and won a Duke's Choice Award in 2013.

The JUG regularly cooperates with other JUGs (such as the Virtual JUG) and supports two smaller local JUGs: the Amsterdam JUG and the Utrecht JUG. It is also a part of the Devoxx4Kids initiative: multiple events organized by its business partners every year that enable kids to learn to code and experiment with technology.

NLJUG members frequently speak at events across Europe and the United States. The NLJUG is always looking for new members to join and help continue the mission of making the Netherlands a great place to be a Java developer. For more information, visit nljug.org.

JOSH JUNEAU

What's New in JPA 2.2

Streaming results, better date conversions, and new annotations are just a few of the many handy improvements.

The Java Persistence API (JPA) is a foundational Java EE specification that is widely used throughout the industry. Whether you are developing for the Java EE platform or an alternative framework for Java, JPA is likely your principal choice for data persistence. JPA 2.1 took the specification to new frontiers, because it enabled developers to perform tasks such as automatically generating a database schema and efficiently working with database stored procedures. The latest release, JPA 2.2, builds upon these improvements and makes incremental improvements to the specification.

In this article, I outline the new features, providing examples that can be used to get started. I use an example project known as the "Java EE 8 Playground," which resides on GitHub. The example application is built upon the Java EE 8 specifications, and it uses the JavaServer Faces (JSF) framework along with Enterprise JavaBeans (EJB) and JPA for persistence. To follow along, you'll need to be familiar with JPA.

Using JPA 2.2

The JPA 2.2 release is part of the Java EE 8 platform. That said, only Java EE 8–compliant application servers provide the specification for use out of the box. At the time of this writing (late 2017), there aren't many Java EE 8–compliant application servers. However, it is still easy to use JPA 2.2 if you're using Java EE 7. The first step is to download the pertinent JAR files using Maven Central and include them with the project. If you are using Maven for the project, add the coordinates to the Maven project object model (POM) file:

```
<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
</dependency>
```

Next, choose the JPA implementation that you want to use. As of the release of JPA 2.2, both EclipseLink and Hibernate have compatible implementations. For the examples in this article, I use EclipseLink by adding the following dependency:

```
<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>eclipselink</artifactId>
    <version>2.7.0 </version>
</dependency>
```

If you are using a Java EE 8–compliant server, such as GlassFish 5 or Payara 5, you should be able to specify a scope of "provided" for these dependencies in the POM file. Otherwise, specify the "compile" scope to include them in the project build.

Java 8 Date and Time Support

Perhaps one of the most welcome new additions is support for the Java 8 Date and Time API. Since the release of Java SE 8 in 2014, developers have been using workarounds to enable the use of the Date and Time API with JPA. Although most workarounds are quite simple, the requirement for basic support of the updated Date and Time API has been long overdue. The JPA support for the Date and Time API includes the following common types:

- `java.time.LocalDate`
- `java.time.LocalTime`
- `java.time.LocalDateTime`

- java.time.OffsetTime
- java.time.OffsetDateTime

I put this into context by first explaining how to support the Date and Time API without JPA 2.2. Because JPA 2.1 is capable only of working with older date constructs such as java.util.Date and java.sql.Timestamp, a converter must be used to convert from data that is stored in a database to older date constructs supported by the JPA 2.1 release and then to the updated Date and Time API for use in an application. A date converter in JPA 2.1 that is capable of performing such conversion might look like that in **Listing 1**. The converter in the listing is used to convert between LocalDate and java.util.Date.

Listing 1.

```
@Converter(autoApply = true)
public class LocalDateTimeConverter implements
     AttributeConverter<LocalDate, Date> {
  @Override
  public Date convertToDatabaseColumn(
              LocalDate entityValue) {
      LocalTime time = LocalTime.now();
      Instant instant = time.atDate(entityValue)
        .atZone(ZoneId.systemDefault())
        .toInstant();
      return Date.from(instant);
    }

    @Override
    public LocalDate convertToEntityAttribute(
              Date databaseValue){
      Instant instant = Instant
        .ofEpochMilli(databaseValue.getTime());
      return LocalDateTime.ofInstant(instant,
        ZoneId.systemDefault()).toLocalDate();
    }
}
```

In JPA 2.2, there is no longer a need for coding such a converter when you are using the supported date-time types. Support for those types is built in, so you can simply specify the supported type on a field of an entity class without any further code. The following code excerpt demonstrates this concept. In the code, note that there is no need to add a `@Temporal` annotation, because the type mapping occurs automatically.

```
public class Job implements Serializable {
 . . .
    @Column(name = "WORK_DATE")
    private LocalDate workDate;
 . . .
}
```

Because the supported date-time types are first-class citizens of JPA, they can simply be specified without the extra ceremony. In JPA 2.1, the `@Temporal` annotation must be specified on all persistent fields or properties of type `java.util.Date` and `java.util.Calendar`.

Although only a subset of the date-time types is supported in this release, an attribute converter can easily be generated for working with other types, such as conversion between `LocalDateTime` and `ZonedDateTime`. The biggest challenge to writing such a converter is determining how to best make the conversion between the different types. To make things even easier, attribute converters are now injectable. I show an example of injection in the next section.

The JPA 2.2 update adds the useful ability of enabling attribute converters to become injectable.

The code in **Listing 2** demonstrates how to convert between `LocalDateTime` and `ZonedDateTime`.

Listing 2.

```
@Converter
public class LocalToZonedConverter implements
 AttributeConverter<ZonedDateTime, LocalDateTime> {
    @Override
    public LocalDateTime convertToDatabaseColumn(
        ZonedDateTime entityValue) {
          return entityValue.toLocalDateTime();
    }

    @Override
    public ZonedDateTime convertToEntityAttribute(
        LocalDateTime databaseValue) {
          return ZonedDateTime.of(databaseValue,
                          ZoneId.systemDefault());
    }
}
```

This particular example is straightforward, because `ZonedDateTime` contains easy-to-use conversion methods. It is converted by calling upon its `toLocalDateTime()` method. The opposite conversion can be made by calling upon the `ZonedDateTimeOf()` method and passing the `LocalDateTime` value along with the `ZoneId` for the time zone to use.

Injectable Attribute Converters

Attribute converters were a very useful addition to JPA 2.1, because they allowed entity attribute types to become much more flexible. The JPA 2.2 update adds the useful ability of enabling attribute converters to become injectable. That is, you can now inject Contexts and Dependency Injection (CDI) resources into an attribute converter. This modification falls in line with other

CDI improvements made throughout the Java EE 8 specifications such as the enhanced JSF converters, because they can also now use CDI injection.

To use this new feature, simply inject CDI resources into an attribute converter, as needed. Listing 2 demonstrates an example of an attribute converter, and now I will walk through it to point out the important pieces.

The converter class must implement the `javax.persistence.AttributeConverter` interface, passing an X and Y value. The X value should correspond to the data type in the Java object, and the Y value should correspond to the database column type. Next, the converter class should be annotated with `@Converter`. Finally, the class should override the `convertToDatabaseColumn()` and `convertToEntityAttribute()` methods. The implementation in each of these methods should convert the values to and from the specified types.

The scrollable ResultSet and pagination techniques fetch only a portion of the data at one time, which might be better with large datasets.

To automatically apply the converter each time the specified data type is used, specify "automatic," as in `@Converter(autoApply=true)`. To apply the converter to a single attribute, use the `@Converter` annotation at the attribute level, as shown here:

```
@Convert(converter=LocalDateConverter.java)
private LocalDate workDate;
```

A converter can also be applied at the class level, as follows:

```
@Convert(attributeName="workDate",
        converter = LocalDateConverter.class)
public class Job implements Serializable {
 . . .
```

Suppose that I wanted to encrypt the values that are contained in the `creditLimit` field of a `Customer` entity when it is persisted. To implement such a process, the values would need to be encrypted prior to being persisted and unencrypted upon retrieval from the database. This can be done within a converter and, by using JPA 2.2, I can inject an encryption object into the converter to achieve the desired result. **Listing 3** provides an example.

Listing3.

```
@Converter
public class CreditLimitConverter implements
    AttributeConverter<BigDecimal, BigDecimal> {

    @Inject
    CreditLimitEncryptor encryptor;

    @Override
    public BigDecimal convertToDatabaseColumn
                      (BigDecimal entityValue) {
        String encryptedFormat =
         encryptor.base64encode(
                entityValue.toString());
        return BigDecimal.valueOf(
         Long.valueOf(encryptedFormat));
    }

    . . .
}
```

In this code, the process is performed by injecting the `CreditLimitEncryptor` class into the converter and making use of it to assist in the process.

Streaming Results of Query Executions

It is now very easy to take full advantage of the Java SE 8 stream features when you are working with the results of query executions. Not only do streams make code easier to read, write, and maintain, but they also can help queries to perform better in some situations. Some stream implementations can also help to prevent querying too much data at once, although in some cases the use of ResultSet pagination can perform better than streams.

To enable this feature, the getResultStream() method has been added to both the Query and TypedQuery interfaces. This minor change allows JPA to simply return a stream of results, rather than a list. Therefore, if you are working with a large ResultSet, it makes sense to do some performance comparisons between the new stream implementation and scrollable ResultSets or pagination. The reason is that the stream implementations will fetch all records at once, storing them in a list and then returning. The scrollable ResultSet and pagination techniques fetch only a portion of the data at one time, which might be better with large datasets.

Be aware of performance in scenarios where lots of data is returned.

Persistence providers can choose to override the new getResultStream() method with a better implementation. Hibernate already includes a stream() method, which uses a scrollable ResultSet to parse through result records rather than returning everything. This enables Hibernate to work with very large datasets and perform very well. Expect other providers to override this method to provide similar features that will be beneficial for JPA.

Performance aside, the option to stream results very easily is a favorable addition to JPA, providing a convenient way to work with data. I am going to demonstrate a couple of the scenarios where this might be a benefit, although there are countless possibilities. In both scenarios, I query the Job entity and return a stream. First, take a look at the following code where I simply parse a stream of Jobs for a specified Customer by calling the Query interface's getResultStream() method. I then use that stream to print out details regarding the Job customer and work date.

```
public void findByCustomer(PoolCustomer customer){
  Stream<Job> jobList =  em.createQuery(
            "select object(o) from Job o " +
    "where o.customer = :customer")
     .getResultStream();
  jobList.map(j -> j.getCustomerId()
    .getCustomerId().getCustomerId()
    + " ordered job " + j.getId()
    + " - Starting " + j.getWorkDate())
    .forEach(jm -> System.out.println(jm));
}
```

This method can be modified slightly to return a list of results by using the `Collectors.toList()` utility method, as follows.

```
public List<Job> findByCustomer(
           PoolCustomer customer){
    Stream<Job> jobList =  em.createQuery(
      "select object(o) from Job o " +
      "where o.customerId = :customer")
      .setParameter("customer", customer)
      .getResultStream();
    return jobList.collect(Collectors.toList());
}
```

In the next scenario, shown below, I find a `List` of jobs that pertain to pools of a particular shape. In this instance, I return all jobs that match the shape passed in as a string. Similar to the first example, I first return a stream of `Job` records. Next, I filter the records based on the customer's pool shape. As can be seen, the resulting code is concise and very easy to read.

```
public List<Job> findByCustPoolShape(
        String poolShape){
    Stream<Job> jobstream = em.createQuery(
      "select object(o) from Job o")
      .getResultStream();
    return jobstream.filter(
      c -> poolShape.equals(c.getCustomerId()
           .getPoolId().getShape()))
        .collect(Collectors.toList());

}
```

As I initially mentioned, it is important to be aware of performance in scenarios where lots of data is returned. There are also circumstances in which streams are beneficial in querying databases and others where they can cause performance degradation. A good rule of thumb is that if the data can be queried within the confines of a SQL query, it might make most sense to do just that. Sometimes the benefits of using the elegant stream syntax do not outweigh better performance that can be gained using standard SQL filtering.

Repeatable annotations became possible, letting the same annotation be repeated on a declaration more than once.

Repeatable Annotation Support

When Java SE 8 was released, repeatable annotations became possible, letting the same annotation be repeated on a declaration more than once. Some situations require the use of the same annotation more than one time on a class or field. For instance, there might be more than one `@SqlResultSetMapping` annotation on a given entity class. In situations such as these, a container annotation had to be used prior to repeatable annotation support. Not only do repeatable annotations reduce the requirement to wrap a collection of the same annotations inside a container annotation, but they can also make code easier to read.

Here's how they work: Behind the scenes, an annotation class implementation must be marked with the @Repeatable meta-annotation to indicate that it can be utilized more than once. The @Repeatable meta-annotation accepts the class type of the container annotation. For example, the NamedQuery annotation class is now marked with the @Repeatable(NamedQueries.class) annotation. In this way, the container annotation is still used, but you do not have to think about it when using the same annotation on a declaration or class because @Repeatable abstracts that detail away.

Here's an example. If you wanted to add more than one @NamedQuery annotation to an entity class in JPA 2.1, you had to encapsulate them inside the @NamedQueries annotation, as seen in Listing 4.

Listing 4.

```
@Entity
@Table(name = "CUSTOMER")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Customer.findAll",
 query = "SELECT c FROM Customer c")
    , @NamedQuery(name = "Customer.findByCustomerId",
 query =
    "SELECT c FROM Customer c "
    + "WHERE c.customerId = :customerId")
    , @NamedQuery(name = "Customer.findByName",
 query = "SELECT c FROM Customer c "
    + "WHERE c.name = :name")
. . .)})
public class Customer implements Serializable {
. . .
}
```

However, in JPA 2.2 this is no longer the case. Because `@NamedQuery` is now a repeatable annotation, it can be listed more than once in an entity class, as shown in **Listing 5**.

Listing 5.

```
@Entity
@Table(name = "CUSTOMER")
@XmlRootElement
@NamedQuery(name = "Customer.findAll",
  query = "SELECT c FROM Customer c")
@NamedQuery(name = "Customer.findByCustomerId",
  query = "SELECT c FROM Customer c "
   + "WHERE c.customerId = :customerId")
@NamedQuery(name = "Customer.findByName",
  query = "SELECT c FROM Customer c "
   + "WHERE c.name = :name")
. . .
public class Customer implements Serializable {
. . .
}
```

Table 1, on the next page, shows the repeatable annotations.

Conclusion

The JPA 2.2 release does not contain a large number of changes, but the enhancements it does include are significant. At last, JPA is brought into alignment with Java SE 8, allowing developers to make use of features such as the Date and Time API, streaming query results, and repeatable annotations. This release also brings better alignment with CDI via the added capability to inject CDI resources into attribute converters. Now that JPA 2.2 is part of Java EE 8 and it is available, I think you'll enjoy using it. </article>

THE FOLLOWING ANNOTATIONS HAVE BEEN MADE REPEATABLE BY THE JPA 2.2 RELEASE:
ASSOCIATIONOVERRIDE
ATTRIBUTEOVERRIDE
CONVERT
JOINCOLUMN
MAPKEYJOINCOLUMN
NAMEDENTITYGRAPH
NAMEDNATIVEQUERY
NAMEDQUERY
NAMEDSTOREDPROCEDUREQUERY
PERSISTENCECONTEXT
PERSISTENCEUNIT
PRIMARYKEYJOINCOLUMN
SECONDARYTABLE
SQLRESULTSETMAPPING

Table 1.

Josh Juneau works as an application developer, system analyst, and database administrator. He primarily develops using Java and other JVM languages. He is a frequent contributor to Oracle Technology Network and *Java Magazine*. Juneau has written several books on Java and Java EE for Apress, and he was a member of the JCP Expert Group for JSR 372 and JSR 378. He is a member of the NetBeans Dream Team, a Java Champion, and a regular voice on the *Java Off Heap* podcast.

MERT ÇALIŞKAN

Eclipse MicroProfile: The Light Enterprise Java Option

Examining a Java EE architecture designed for microservices and distributed applications

Eclipse MicroProfile defines a reduced-size platform for developing microservices using the power of Java EE. MicroProfile 1.0 was first released at JavaOne 2016 with four different runtimes coming from these vendors: IBM, Payara, Tomitribe, and Red Hat. The names of the runtimes were

- IBM WebSphere Liberty
- Payara MicroProfile
- TomEE (from Tomitribe)
- WildFly Swarm (from Red Hat)

All runtimes contain the three core Java EE specifications, which are JAX-RS 2.0, Contexts and Dependency Injection (CDI) 1.2, and JSON-P 1.0. The runtime is defined to be a very small subset of the Java EE specifications, because the driver behind MicroProfile is to provide a foundation for building microservices-based architectures. This article assumes that you are familiar with these specifications and have a basic understanding of them.

MicroProfile 1.1 was released in August of this year, and it contains the final version of MicroProfile-Config 1.0. I'll cover more on that in the following section.

MicroProfile 1.2 was released just before this issue went to press, and version 2.0 is scheduled for next year. MicroProfile 1.2 aligns with specifications from Java EE 7, and version 2.0 will align with specifications from Java EE 8.

Eclipse MicroProfile currently focuses on five foundational concepts: configuration, health check, fault tolerance, metrics, and security. Taken together, these features should make a microservice implementation configurable, resilient, robust, and secure. In this article, I provide usage details for the APIs corresponding to these specifications. I expect you'll like how well the services address microservices issues and how straightforward the APIs are.

MicroProfile-Config

The MicroProfile-Config specification provides the ability to separate the implementation from its configuration so that deployments of an artifact can behave differently. It also enables modification of the configuration of a microservice, for instance, in a dynamic way without the need of repackaging or redeploying artifacts.

The MicroProfile-Config feature achieves this by aggregating the configurations from many different sources—such as system properties, environment variables, and the `META-INF/microprofile-config.properties` file or from custom `ConfigSource` implementations—and presents the complete configuration as a single, merged view to the user.

The current available version of the API is 1.1. Its artifact can be retrieved via Maven, as shown in **Listing 1**, with the dependency definition. The artifact is already available in the central Maven repository. It's also worth mentioning that this API requires at least Java 8 in order to run. Here is how to include it in a Maven-based project:

■ **Listing 1.**

```
<dependency>
   <groupId>org.eclipse.microprofile.config</groupId>
   <artifactId>microprofile-config-api</artifactId>
   <version>1.1</version>
</dependency>
<repositories>
```

Keep in mind that this refers just to the API, and it will be backed up by an implementation coming from the MicroProfile runtimes distributed via vendors. Let's continue with code samples on how the API can be used.

Configurations can be easily injected with annotations, and I'll show how a little later. But they can also be obtained programmatically, as shown in Listing 2.

Listing 2.

```
public class ConfigUsageExample {

    public void useTheConfig() {
        Config config = ConfigProvider.getConfig();

        String baseUrl =
            config.getValue("server.base.url", String.class);

        invokeEndpoint(baseUrl);
    }
}
```

This configuration can be overridden easily at runtime via a system property. For example, see the snippet given in Listing 3, which overrides `server.base.url`:

Listing 3.

```
java -jar myservice.jar \
     -Dserver.base.url=http://another.server/endpoint
```

The configuration can also be accessed via injection into your beans by using `@Inject` and `@ConfigProperty` annotations, as shown in Listing 4.

Listing 4.

```
@ApplicationScoped
public class InjectedConfigUsageExample {

    @Inject
    @ConfigProperty(name="server.base.url")
    private String baseUrl;

    @Inject
    @ConfigProperty(name="server.port")
    private Optional<Integer> port;
}
```

The given property, `server.base.url`, must be defined in one of the provided configuration sources; a `DeploymentException` will be thrown if no definition exists. On the other hand, the `server.port` property is defined with `Optional`, so a `DeploymentException` will not occur if the property is missing in the configuration.

MicroProfile-FaultTolerance

The MicroProfile-FaultTolerance specification provides strategies for implementing resilient applications. Its aim is to separate the execution logic from the execution itself by focusing on features such as TimeOut, RetryPolicy, Fallback, CircuitBreaker, and Bulkhead. I will describe each concept with code samples.

Timeout. This feature prevents the execution of a microservice from being unresponsive. It's vital to have a timeout value defined for a microservice, so that an alternative execution mechanism can be applied (such as Fallback, RetryPolicy, CircuitBreaker, and so forth).

Methods annotated with `@Timeout`, as shown in **Listing 5**, will have the policy applied, and if execution exceeds the given timeout value, a `TimeoutException` will be thrown. Annotations can be used at the class level as well, in which case the timeout policy is applied to all methods defined within that class.

Listing 5.

```
@Timeout(500) // timeout is 500 ms
public List<Recommendation> fetchRecommendations() {
    // Aggregate recommendations
}
```

It's recommended that you use `@Timeout` together with `@Asynchronous` so that the microservice can be executed in a separate thread and can therefore finish at any time.

RetryPolicy. This feature provides a retry policy to overcome problems with connectivity. The `@Retry` annotation shown in Listing 6 can be applied at the method level or the class level. Method-level definitions will override class-level definitions if both exist. In the given example, the maximum retry count is 3, and the maximum duration is 10,000 milliseconds, which is 10 seconds. When the specified duration is exceeded, no more retries will be performed regardless of the retry count.

Listing 6.

```
@Retry(maxRetries = 3, maxDuration= 10000)
public void invokeService() {
    // invoke an external service
}
```

RetryPolicy can also be set for an exception, as shown in Listing 7. This is a convenient way of handling exceptions while doing work on external resources such as I/O processing.

Listing 7.

```
@Retry(retryOn = {IOException.class})
public void readFile() {
    // ...
}
```

Fallback. This feature provides an alternative way to handle a failed execution. It will be invoked once Timeout, RetryPolicy, or CircuitBreaker has failed according to its contract. Fallback definition for a method restricted with a timeout value and its handler implementation are shown in Listing 8. If the execution of the `service.retrieveAmount()` method takes more than 3 seconds, the handler will be executed and 42 will be returned as the amount.

Listing 8.

```
@Timeout(3000)
@Fallback(MyFallback.class)
Long getAmount() {
    return service.retrieveAmount();
}

public class MyFallback implements FallbackHandler<Long> {
   Long handle(ExecutionContext context) {
      return 42;
   }
}
```

CircuitBreaker. This feature prevents repeating failure scenarios, so that invocation of a microservice fails as fast as possible. **Listing 9** shows an example of a CircuitBreaker definition on a method call. CircuitBreaker can be used with Timeout, Fallback, Asynchronous, Bulkhead, and RetryPolicy.

Listing 9.

```
@CircuitBreaker(successThreshold = 10,
                requestVolumeThreshold = 4,
                failureRatio=0.75,
                delay = 1000)
public void invokeService() {
    // invoke an external service
}
```

This CircuitBreaker specifies that the circuit will open, meaning that calls to the invokeService() method will fail once three failures (requestVolumeThreshold x failureRatio) occur within the last four consecutive invocations. The circuit will stay open for 1 second (delay) and then will transit to a half-open state. This half-open state means that if at least one call to the service fails, the circuit will remain open; otherwise, it needs 10 consecutive calls to succeed (successThreshold) in order to transit to the closed state. The closed state is the default state, meaning that method invocations work as expected with no failures.

Bulkhead. This feature limits the concurrent requests to a microservice in order to prevent a system-wide failure. This can be thought of as a barrier between failing code execution and the rest of the functioning part of the system.

It offers either thread-pool isolation or semaphore isolation. Thread-pool isolation can be done via the @Asynchronous annotation. The example in Listing 10 limits concurrent requests to 3 with a waiting queue size of 5. When the limit is reached, threads will be stored in the waiting queue; and when the waiting queue reaches its limit, BulkheadException will be thrown.

Listing 10.

```
@Asynchronous
@Bulkhead(value = 3, waitingThreadQueue = 5)
public Future<Connection> acquireConnection() {
    Connection conn = createConnection();
    return CompletableFuture.completedFuture(conn);
}
```

An example of the semaphore-style bulkhead definition is shown in Listing 11.

Listing 11.

```
@Bulkhead(value = 3)
public Future<Connection> acquireConnection() {
    Connection conn = createConnection();
    return CompletableFuture.completedFuture(conn);
}
```

MicroProfile-HealthCheck

The MicroProfile-HealthCheck specification provides an API to diagnose a microservice's health, using a *producer*, and take measurements of its state. I hope MicroProfile runtimes will provide procedures for checking the health of the services, so let's look at how a procedure is implemented and the result of execution as the response.

Procedures need to implement the `HealthCheck` interface. They return an instance of `Response` as the result. The implementation details are given in **Listing 12**.

Listing 12.

```
@FunctionalInterface
public interface HealthCheck {
    Response call();
}

public abstract class Response {
    public enum State { UP, DOWN }
    public abstract String getName();
    public abstract State getState();
    public abstract
       Optional<Map<String, Object>> getAttributes();
    ...
}
```

A sample implementation for a HealthCheck service is shown in **Listing 13**.

Listing 13.

```
public class SuccessfulCheck implements HealthCheck {
    @Override
    public Response call() {
        return Response.named("successful-check").up();
    }
}
```

Producers should provide an HTTP endpoint, but they can also support protocols such as TCP or Java Management Extensions (JMX). Requests sent to a producer may be protocol-specific, but the response should be in JSON format mapped with a proper HTTP status code. Status codes with their detailed explanations are shown in Table 1.

A sample JSON response is shown in Listing 14. Key-value pairs can be defined within the data to provide as much as detail possible.

Listing 14.

```
{
  "outcome": "UP",
  "checks": [
    {
      "id": "diskCheck",
      "result": "UP",
      "data": {
        "disk.free.space": "20gb"
      }
    }
  ]
}
```

The MicroProfile-HealthCheck specification can check on the amount of heap being used, excessive garbage collection, running out of disk space, threads that are deadlocked, and so on.

STATUS CODE	DESCRIPTION
200	EXECUTION OF THE CHECK RESULTED IN A POSITIVE OUTCOME
204	NO HEALTH CHECK WAS INSTALLED WITHIN THE RUNTIME
503	EXECUTION OF THE CHECK RESULTED IN A NEGATIVE OUTCOME
500	EXECUTION OF THE CHECK CANNOT BE COMPLETED DUE TO A PROBLEM

Table 1. Status codes

MicroProfile-Security

Security in a microservices-based architecture mostly resembles RESTful service security methodologies. RESTful services are usually stateless; therefore, the security context will be re-created on every request with the provided token from the client. The MicroProfile-Security specification follows this token-based approach by employing JSON Web Tokens (JWTs) for handling authentication and role-based authorization. An example of the JWT token format is shown in Listing 15.

Listing 15.

```
{
  "iss":"https://server.example.com",
  "sub":"24400320",
  "preferred_username":"jdoe",
  "aud":"s6BhdRkqt3",
  "nonce":"n-0S6_WzA2Mj",
  "exp":1311281970,
  "iat":1311280970,
  "auth_time":1311280969,
  "realm_access":{
    "roles":[
      "role-in-realm",
      "user",
      "manager"
    ]
  },
  "resource_access":{
    "my-service":{
      "roles":[
        "role-in-my-service"
      ]
    }
```

The industry is shifting its course from monolithic architectures to more microservices-based models, and new specifications in the Java EE ecosystem are now on the horizon to meet those needs.

```
  }
}
```

In addition to the standard abilities of a JWT, the italicized lines in **Listing 15** are add-ons defined by the specification. The username will be carried by the `preferred_username` claim, and the granted roles of the realm will be carried by the `realm_access` claim. The `resource_access` claim will define the set of roles specific to a service.

Conclusion

The industry is shifting its course from monolithic architectures to more microservices-based models, and new specifications in the Java EE ecosystem are now on the horizon to meet those needs. Java EE has gone through a long path of releases in its 20 years of life. This has made it mature enough to be the foundation for microservices and for the new specifications currently emerging. Eclipse MicroProfile uses core Java EE specifications, such as CDI and JAX-RS, as a foundation to build emerging solutions for requirements that are shaping the microservices ecosystem. The four specifications detailed in this article are just a beginning; additional specifications have been drafted in the pipeline of the working group, and you'll be hearing more about them in the near future. </article>

[Update: As we went to press, Oracle announced that it had joined the MicroProfile initiative. —*Ed.*]

Mert Çalişkan (@mertcal) is a Java Champion, a coauthor of *PrimeFaces Cookbook* (Packt Publishing, 2013) and *Beginning Spring* (Wiley Publications, 2015), and a frequent contributor to *Java Magazine*. In addition, he actively contributes to the Ankara JUG, the largest Java user group in Turkey. Çalişkan works as a developer on Payara Server inside the Payara Foundation.

BEN EVANS

Understanding Java Method Invocation with Invokedynamic

The instruction added in Java 7 makes it possible to resolve method calls dynamically at runtime.

In the first part of this two-part series, I discussed four of Java's five method-invocation opcodes. These are the bytecode representations of the standard forms of method invocation used in Java 8 and Java 9.

This raises the question of how the fifth opcode, invokedynamic, enters the picture. The short answer is that, as of Java 9, there is no direct support for invokedynamic in the Java language. In fact, when invokedynamic was added to the runtime in Java 7, the javac compiler would not emit the new bytecode under any circumstances whatsoever.

As of Java 8, invokedynamic is used as a primary implementation mechanism to provide advanced platform features. One of the clearest and simplest examples of this use of the opcode is in the implementation of lambda expressions. To follow along with the rest of this article, you'll need to have some familiarity with how the JVM invokes methods, or you'll need to read the first article in this series.

Lambdas Are Object References

Before diving into how invokedynamic is used to enable lambdas, a brief reminder of what lambdas actually are is in order. Java has only two types of values: primitive types (such as char, int, and so on) and object references. Lambdas are obviously not primitive types, so they must be object references. Consider this lambda:

PHOTOGRAPH BY JOHN BLYTHE

```
public class LambdaExample {
    private static final String HELLO = "Hello";

    public static void main(String[] args) throws Exception {
        Runnable r = () -> System.out.println(HELLO);
        Thread t = new Thread(r);
        t.start();
        t.join();

    }
}
```

The lambda expression on line 5 is assigned to a variable of type Runnable. This means that the lambda evaluates to a reference to an object that has a type that is compatible with Runnable. Essentially, this object's type will be some subclass of Object that has defined one extra method (and has no fields). The extra method is understood to be the run() method expected by the Runnable interface.

Before Java 8, such an object was represented only by an instance of a concrete anonymous class that implemented Runnable. In fact, in the initial prototypes of Java 8 lambdas, inner classes were used as the implementation technology.

The long-range future roadmap for the JVM could contain future versions where more-sophisticated representations of lambdas could be possible. Fixing the representation to use explicit inner classes would prevent a different representation being used by a future version of the platform. This is undesirable and so, instead, Java 8 and Java 9 use a more sophisticated technique than hardcoding inner classes. The bytecode for the previous lambda example is as follows:

```
public static void main(java.lang.String[]) throws java.lang.Exception;
    Code:
       0: invokedynamic #2,  0 // InvokeDynamic
                               // #0:run:()Ljava/lang/Runnable;
```

```
 5: astore_1
 6: new           #3     // class java/lang/Thread
 9: dup
10: aload_1
11: invokespecial #4     // Method java/lang/Thread."<init>":
                         // (Ljava/lang/Runnable;)V
14: astore_2
15: aload_2
16: invokevirtual #5     // Method java/lang/Thread.start:()V
19: aload_2
20: invokevirtual #6     // Method java/lang/Thread.join:()V
23: return
```

The bytecode at offset 0 indicates that some method is being called via invokedynamic, and the return value of that call is placed upon the stack. The rest of the bytecode in the method is a straightforward representation of the rest of the method.

How Invokedynamic Operates

At this point, I should discuss some of the details of the nature of invokedynamic and how the opcode operates. When a class containing an invokedynamic instruction is loaded by the class loader, the target of the method invocation is not known ahead of time. This design differs from all other types of call sites in JVM bytecode.

For example, in the case of invokestatic and invokespecial sites, which I discussed in the previous article, the exact implementation method (referred to as the *call target*) is known at compile time. In the case of invokevirtual and invokeinterface, the call target is determined at runtime. However, the target selection is subject to the constraints of the Java language inheritance rules and type system. As a result, at least some call target information is known at compile time.

In contrast, invokedynamic is far more flexible about which method will actually be called when the opcode is dispatched. To allow for this flexibility, invokedynamic opcodes refer to a special attribute in the constant pool of the class that contains the dynamic invocation. This attribute contains additional information to support the dynamic nature of the call, called bootstrap methods (BSMs). These are a key part of invokedynamic, and every invokedynamic call site has a constant pool entry for a corresponding BSM. To allow the association of a BSM to a specific invokedynamic call site, a new entry type, also called *InvokeDynamic*, has been added to the class file format as of Java 7.

The call site of the invokedynamic instruction is said to be "unlaced" at class loading time. The BSM is called to determine what method should actually be called, and the resulting `CallSite` object will then be "laced" into the call site.

In the simplest case, that of a `ConstantCallSite`, as soon as the lookup has been done once, it will not need to be repeated. Instead, the target of the call site will be directly called on all future invocations without any further work. This means that the call site is now stable and is, therefore, friendly to other JVM subsystems, such as the just-in-time (JIT) compiler.

One important difference between method handles and reflection is that lookup contexts return only methods that were accessible from the scope where the lookup object was created, which means they are safe to use under all circumstances.

For this mechanism to work efficiently, the JDK must contain suitable types to represent the call site, the BSMs, and other parts of the implementation. Java's original core reflection types are capable of representing methods and types. However, the API dates from the very early days of the Java platform and has several aspects that make it a less-than-ideal choice.

For example, reflection predates both collections and generics. As a result, method signatures are represented by `Class[]` in the Reflection API. This can be cumbersome and error-prone, and it is hampered by the verbose nature of Java's array syntax. It is further complicated by the need to manually box and unbox primitive types and to work around the possibility of void methods.

Method Handles to the Rescue

Instead of forcing the programmer to deal with these issues, Java 7 introduced a new API, called MethodHandles, to represent the necessary abstractions. The core of this API is the package java.lang.invoke and especially the class MethodHandle. Instances of this type provide the ability to call a method, and they are directly executable. They are dynamically typed according to their parameter and return types, which provides as much type safety as possible, given the dynamic way in which they are used. The API is needed for invokedynamic, but it can also be used alone, in which case it can be considered a modern, safe alternative to reflection.

To get a handle for a method, the method must be looked up via a *lookup context*. The usual way to get a context is to call the static helper method MethodHandles.lookup(). This method returns a lookup context based on the currently executing method. From this context, you can obtain method handles by calling one of the find*() methods (for example, findVirtual() or findConstructor()).

You can think of the invokedynamic opcode as representing a call to some sort of platform factory method for a lambda expression.

One important difference between method handles and reflection is that lookup contexts return only methods that were accessible from the scope where the lookup object was created. There is no way to subvert this and no equivalent of the setAccessible() back door that is present in reflection. This means that method handles are safe to use under all circumstances, including using them with a security manager.

However, care must be taken, as the access control check has been moved to method-lookup time. This means that a lookup context can hand out references to private methods that were visible to the lookup, but are not necessarily visible at the time when the method handle is invoked.

In order to solve the problems of representing method signatures, the MethodHandles API also includes the MethodType class. This is a simple immutable type with some very useful properties. It does the following:

- Represents the type signature of a method
- Consists of the return type followed by the argument types
- Does not include the "receiver type" or name of the method
- Is designed to remove the Class[] problem from core reflection

In addition, instances of it are immutable.

With this API, signatures of methods are represented as instances of MethodType, and there is no need to create a new type to model each possible signature. New instances are created from a simple factory method:

```
// toString()
MethodType mtToString =
    MethodType.methodType(String.class);

// A setter method
MethodType mtSetter =
    MethodType.methodType(void.class, Object.class);

// compare() from Comparator<String>
MethodType mtStringComparator =
    MethodType.methodType(int.class, String.class, String.class);
```

Once you have created a signature object, it can be used (along with a method name) to look up a method handle, as in the following example to get a method handle on toString().

```
public MethodHandle getToStringHandle() {
    MethodHandle mh = null;
    MethodType mt = MethodType.methodType(String.class);
    MethodHandles.Lookup lk = MethodHandles.lookup();

    try {
        mh = lk.findVirtual(getClass(), "toString", mt);
    } catch (NoSuchMethodException | IllegalAccessException mhx) {
        throw new AssertionError().initCause(mhx);
    }

    return mh;
}
```

The handle can then be invoked in a similar way to a reflective call. A receiver object must be supplied for instance methods, and the invocation code must deal with the possibility of a coarse-grained exception.

```
MethodHandle mh = getToStringMH();
try {
    mh.invoke(this, null);
} catch (Throwable e) {
    e.printStackTrace();
}
```

The concept of a BSM should now be clear: when program control reaches an invokedynamic call site for the first time, the associated BSM is called. The BSM returns a call site object containing a method handle to the method that will actually be bound into the call site. For this mechanism to function correctly with static typing, the BSM must return a handle to a method of the correct method signature.

To get back to the lambda expression example I gave earlier, you can think of the invokedynamic opcode as representing a call to some sort of platform factory method for a lambda expression. The actual body of the lambda has been transformed into a private static method on the class where the lambda is defined.

```
  private static void lambda$main$0();
    Code:
       0: getstatic     #7  // Field
                            //  java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #9  // String Hello
       5: invokevirtual #10 // Method
                            //  java/io/PrintStream.println:
                            //  (Ljava/lang/String;)V
       8: return
```

The lambda factory will return an instance of some type that implements `Runnable`, and the `run()` method of that type will call back to this private method when the lambda is executed.

Using `javap -v` to look inside the constant pool shows this entry:

```
    #2 = InvokeDynamic      #0:#40         //
#0:run:()Ljava/lang/Runnable;
```

Looking at the BSM section of the class file shows the factory that is being called:

```
BootstrapMethods:
  0: #37 REF_invokeStatic
java/lang/invoke/LambdaMetafactory.metafactory:(Ljava/lang/invoke/MethodH
andles$Lookup;Ljava/lang/String;Ljava/lang/invoke/MethodType;Ljava/lang/i
nvoke/MethodType;Ljava/lang/invoke/MethodHandle;Ljava/lang/invoke/MethodT
ype;)Ljava/lang/invoke/CallSite;
    Method arguments:
      #38 ()V
      #39 REF_invokeStatic optjava/LambdaExample.lambda$main$0:()V
      #38 ()V
```

This output refers to a static factory method, called `metafactory()`, on the `LambdaMetafactory` implementation class in `java.lang.invoke`. This is a BSM that will create the linkage bytecode at runtime, if the lambda is ever created. The metafactory code takes in a lookup object and the method types to ensure static type safety, along with a method handle pointing at the private static method containing the lambda method body. It returns a callsite that will “lace in” the lambda body if it is ever called.

```
public static CallSite metafactory(
        MethodHandles.Lookup caller,
        String invokedName,
        MethodType invokedType,
        MethodType samMethodType,
        MethodHandle implMethod,
        MethodType instantiatedMethodType)
            throws LambdaConversionException {
    AbstractValidatingLambdaMetafactory mf;
    mf = new InnerClassLambdaMetafactory(
            caller, invokedType,
            invokedName, samMethodType,
            implMethod, instantiatedMethodType,
            false, EMPTY_CLASS_ARRAY, EMPTY_MT_ARRAY);
    mf.validateMetafactoryArgs();
    return mf.buildCallSite();
}
```

The current implementation uses a private metafactory that will still create an inner class per lambda, but the classes are dynamically created and are never written to disk. This means that the implementation mechanism could change with a future release of Java, and any existing lambdas will be able to take advantage of the new mechanism.

In Java 8 and Java 9, the implementation based on the `InnerClassLambdaMetafactory` class makes use of a slightly modified version of the ASM bytecode manipulation library that ships in the package `jdk.internal.org.objectweb.asm`.

This implementation creates dynamic classes to represent the implementing type of a lambda, while at the same time future-proofing the implementation and maintaining JIT-friendliness.

It makes use of the simplest case—call sites that are looked up once and cannot change thereafter. These are represented by instances of `ConstantCallSite`, which I discussed earlier. More-complex cases are possible, including call sites that can change or even have semantics similar to volatile variables. These cases are harder to handle and quickly become very complex, but they provide the greatest amount of dynamic flexibility available to the platform.

The previous example of lambda expressions shows how the invokedynamic opcode relaxes a key part of the static type system and makes flexible runtime dispatch possible.

Conclusion

While invokedynamic might not be a part of Java that most developers are exposed to very often, the Java ecosystem has evolved significantly through its addition. Future versions of Java may well introduce further advances in VM technology, and many of these techniques would be impossible without the advent of invokedynamic and the reimagining of method execution that it represents. </article>

Ben Evans (@kittylyst) is a Java Champion, tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee.

learn more

Demystifying invokedynamic, Part 1, by Julien Ponge; *Java Magazine*, January/February 2013 (PDF)

Demystifying invokedynamic, Part 2, by Julien Ponge; *Java Magazine*, May/June 2013 (PDF)

NICOLAS PONSINI

FLORIAN TOURNIER

An Introduction to Java Card

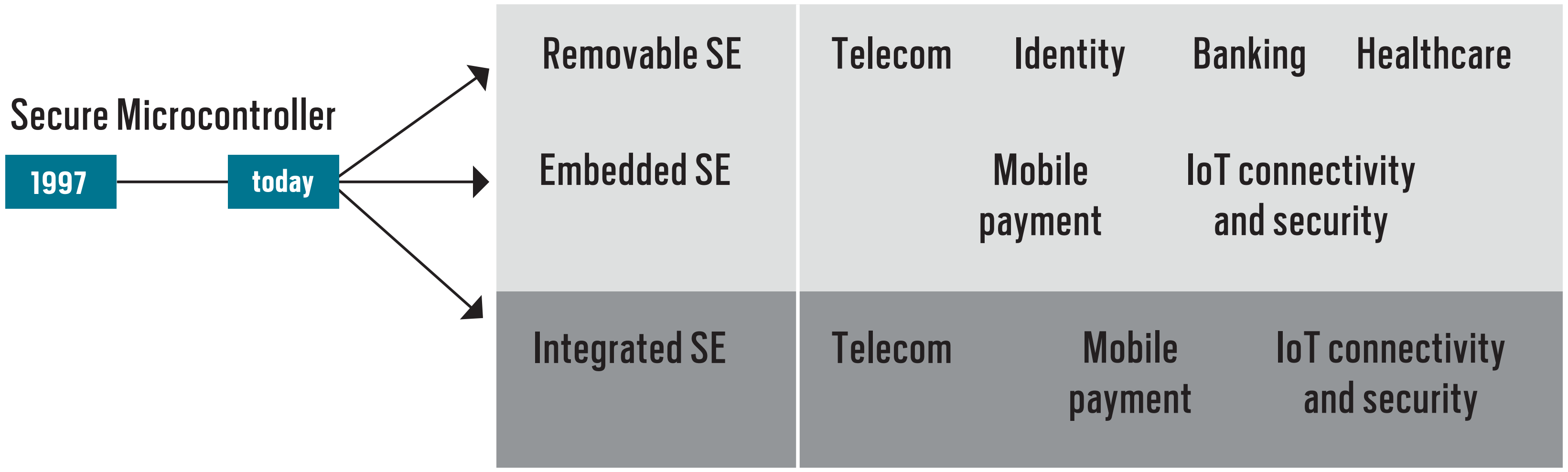
The smallest Java platform is one of the most widely distributed. See how programming it is different from developing conventional apps.

Java Card is at its core a minimal subset of Java, enriched with features catering to the security needs of secure elements. A *secure element* (SE) is a tamper-resistant hardware environment capable of securely hosting applications and their confidential and cryptographic data. The most common SE is the one-chip secure microcontroller found in smartcards. New form factors have started to emerge, though, from embedded SEs (a nonremovable secure microcontroller soldered onto a device board) to new security designs embedded into general-purpose chips. Figure 1 shows the kinds of SEs commonly deployed.

For any of these SEs, a set of common, critical requirements can be identified:

- **Security.** Applications must factor in security attributes such as transaction atomicity, cryptography support, signing an authentication of applications, application isolation, and a firewall.
- **Certifiability.** Customers require high-level security certifications according to internal security assurance level standards such as Common Criteria and FIPS, as well as domain-specific certifications, for example, from government entities or payment organizations.
- **Compactness.** SEs are typically resource-constrained devices (CPU, memory, and bandwidth). In particular, memory configurations rarely exceed 1 MB of overall available memory, and RAM specifications can be as low as 2 KB.
- **Standards-based manageability.** SE applications and stored credentials must be securely managed and updated, according to open industry standards.

Java Card addresses these requirements, while retaining the openness and code portability provided by Java.



Other secure hardware options: Trusted execution environments (TEEs), secure multipoint control units (MCUs), software environments

Figure 1. Commonly deployed secure elements

The Java Card Ecosystem

The number of Java Card–based SEs has been growing steadily over the years, and Java Card is now the most pervasive application platform in the world. It is estimated that more than 6 billion Java Card–based SEs are being deployed in 2017. The majority of those are smartcards (SIM cards, banking cards, and ID cards). However, in recent years, there has been strong growth of Java Card–based SEs embedded in mobile devices. New types of SEs are also being introduced in edge devices for Internet of Things (IoT) networks.

Applications using Java Card technology are multiple and vary greatly across vertical markets. Java Card is used for the following, among others:

- Identity, authentication, and access control
- Secure transactions, including contact and contactless payment
- Credential storage and content security
- Subscription management
- Device integrity check and device attestation
- Digital rights management (DRM)

End users of Java Card technology include mobile operators, financial institutions, governments, mobile device makers, healthcare associations, enterprises, and transportation authorities. Standards bodies such at ETSI, GlobalPlatform, ISO, and others leverage Java Card as part of their specifications. Although the Java Card developer community is specialized and relies on expertise that shares some DNA with the rest of Java, it is rapidly expanding. End users are building in-house Java Card development expertise, and academics are increasingly contributing research and open source content.

Platform Evolution

Java Card was introduced in 1996. It was a pioneering concept at the time (it was the first very small Java framework—before Java ME was introduced), and it was also very incomplete (no virtual machine [VM] specification and only a basic cryptography framework). Subsequent versions altered the architecture and augmented the APIs to the point that the modern Java Card framework would be unrecognizable to users of that first version.

A key contributor and promoter of that evolution has been the Java Card Forum (JCF). The JCF was formed shortly after the initial release of Java Card and, according to its website, it is "a collaboration of companies from the smart card, secure operating system, and secure silicon industry, working together to promote and develop Java as the preferred programming language for multi-application smart cards and secure devices."

The JCF provides recommendations to Oracle for the evolution of the Java Card specifications. It has been instrumental in bringing key innovations to the platform. After version 2.0 started specifying the Java Card VM and runtime conditions, version 2.1 brought an interoperable file format. Version 2.2 brought alignment with the European Telecommunications Standards Institute (ETSI) and enabled contactless smartcards to be supported. Version 3.0 introduced two variants of the specification (Classic and Connected), as well as broader cryptography support. The JCF is now working with Oracle toward the definition of version 3.1, which will enable new use cases in the IoT space. It is slated for release in 2018.

Inside Java Card

Oracle provides a wide range of components to develop Java Card applications, including specifications, development tools, and security documentation.

Java Card specification. Figure 2 shows the Java Card development platform. As you can see, the Java Card specification contains three primary components that support application development.

- The VM specification for the Java Card platform provides the instruction set of the Java Card VM, the supported subset of the Java language, and the file formats used to install applets and libraries into Java Card technology–enabled devices.
- The runtime environment (RE) specification for the Java Card platform defines the necessary behavior of the RE in any implementation of Java Card technology. The RE includes the implementation of the Java Card VM, the Java Card API classes, and runtime support services such as the selection and deselection of applets.
- The API for the Java Card platform complements the Java Card RE specification. It contains the class definitions to support the Java Card VM and the Java Card RE.

Java Card Development Kit. This freely available download includes a complete, standalone development environment in which applications written for the Java Card platform can be

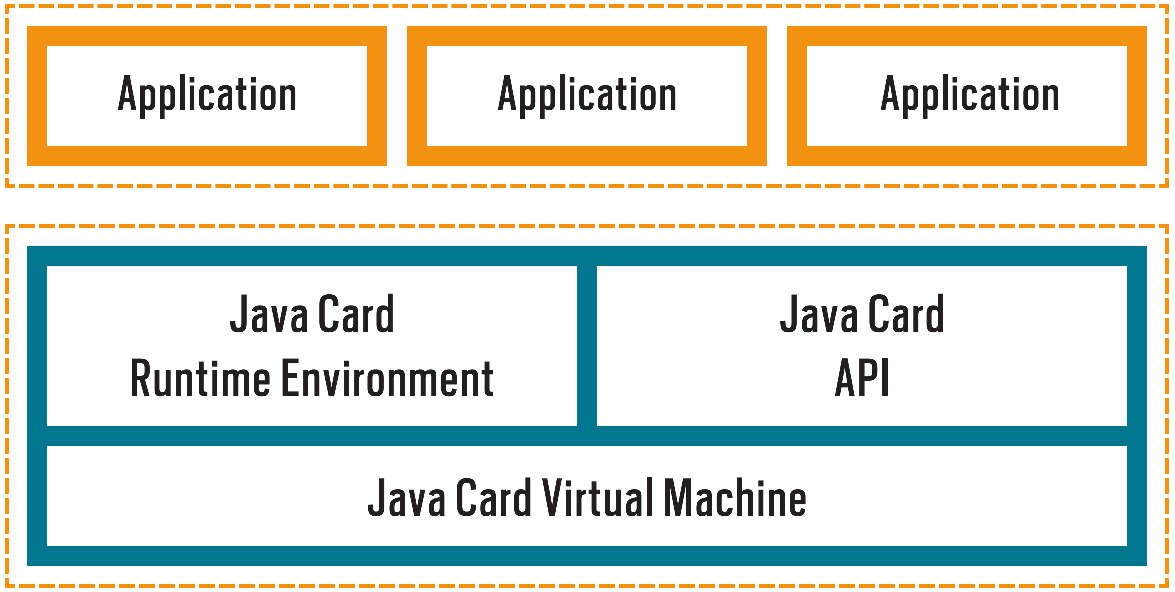


Figure 2. The Java Card development platform

developed. It also includes a complete Java Card simulator and integration with Eclipse, in order to facilitate the testing of applications.

Java Card Protection Profile. The Java Card Protection Profile helps creators of products based on Java Card to meet the security demands of banks, governments, and other card issuers for security evaluations. It provides a modular set of security requirements designed specifically for the characteristics of the Java Card platform. It reduces the time and cost for developers of Java Card–based products to complete security evaluations using the Common Criteria for IT Security Evaluation standard.

What Makes Java Card Development Unique?

There are two clichés to avoid prior to describing some Java Card specificities.

The first is that Java Card is just for smartcards. The word *Card* in Java Card is a misnomer. A more accurate name (but with less marketing appeal) might be "Java for resource-constrained devices to run securely Java technology–based applications." While the Java Card framework and runtime are security-oriented and designed to run on a secure SE with limited memory and countermeasures against hardware attacks, other form factors are equally suited to run Java Card in different markets. For example, Java Card is now the security framework for all kinds of security devices.

The second cliché is that a knowledgeable Java developer is a Java Card developer. Even skilled Java developers might be disconcerted by Java Card programming. While Java Card is a Java subset, it has several additional specificities such as object persistency and atomicity. As a result, Java Card development has historically been the work of a dedicated crowd of specialists. Developing a formally proven applet for long-lasting deployment and protection against software and hardware attacks requires strong security skills. Anyone can play with the Java Card Development Kit (and is encouraged to do so), but it can take time and experience to acquire the necessary expertise to deploy an applet in the field.

The remainder of this article explains key concepts that make Java Card unique compared with other Java environments.

Java Language Subset

Java Card is a subset of the Java language (JDK 7). Next, I list the key supported and unsupported language features.

Java Card–supported JVM features include

- Java's object-oriented features such as classes, interfaces, inheritance, and exceptions as well as packages
- Small primitive data types: `boolean`, `byte`, `short`, `int` (optional)
- One-dimensional arrays
- Some methods of `Object` and `Throwable`
- Classes
- The Garbage collector is optional and might or might not be present.

Java Card unsupported JVM features include

- Dynamic class loading
- The security manager
- Threads
- Cloning
- Large primitive data types (`long`, `double` `float`)
- Characters and strings
- Multidimensional arrays
- Keywords (`native`, `synchronized`, `transient`, `volatile`, and so on)

Java Card API

The following packages are the key libraries in the Java Card API.

`java.io` defines a subset of the `java.io` package that's in the standard Java programming language and consists of the `java.io.IOException` class to maintain a hierarchy of exceptions identical to the standard Java programming language.

`java.lang` provides classes that are fundamental to the design of the Java Card technology subset of the Java programming language. The classes in this package are derived from `java.lang` in the standard Java programming language and represent the core functionality required by the Java Card VM. This core functionality is represented by the Object class, which is the base class for all Java language classes, and the Throwable class, which is the base class for the exception and runtime exception classes.

javacard.framework provides a framework of classes and interfaces for building, communicating with, and working with Java Card technology–based applets. It defines core concepts of Java Card such as the APDU (Application Protocol Data Unit) class, the Applet (Java Card applet) class, the JCSystem (Java Card System) class, the PIN (Personal Identification Number) interface, and various Java Card–specific exceptions.

javacard.security defines a security API that includes various types of keys and algorithms for symmetric (AES) and asymmetric (RSA, ECC, DH, and so on) operations, message digests, and signatures.

javacardx.crypto is an extension package that contains functionality, which may be subject to export controls, for implementing a security and cryptography framework on the Java Card platform. The platform must support this optional package only if cryptographic encryption and decryption functionality is included in the implementation. It contains the Cipher class and the KeyEncryption interface. Cipher provides methods for encrypting and decrypting messages. KeyEncryption provides functionality that allows keys to be updated in a secure end-to-end fashion.

javacardx.biometry and javacardx.biometry1toN are extension packages that contain functionality for implementing biometric frameworks on the Java Card platform.

javacardx.external is an extension package that provides mechanisms to access memory subsystems, which are not directly addressable by the Java Card RE on the Java Card platform.

For the sake of completeness, here is the list of the packages of the Java Card API:

- java.lang (subset)
- java.io (subset)
- javacard.framework
- javacard.framework.service
- javacard.security

- `javacardx.crypto`
- `javacardx.annotations`
- `javacardx.apdu`
- `javacardx.biometry`
- `javacardx.biometry1toN`
- `javacardx.external`
- `javacardx.framework.math`
- `javacardx.framework.string`
- `javacardx.framework.tlv`
- `javacardx.util`
- `javacardx.security`

At the next reset, the VM starts again and recovers its previous heap from persistent storage.

Virtual Machine Lifecycle and Persistence

The Java Card VM and related runtime conditions are unique aspects of the Java Card specifications. The VM is not executed as a process of a host operating system that can be ended. Rather, it is executed forever and its lifetime is that of the host hardware, which means it will never be terminated and its related applications and objects will not be automatically destroyed.

The VM and the created objects are used to represent application information that is persistent and will survive a power loss. It is just stopped at that time; at the next reset, the VM starts again and recovers its previous heap from persistent storage. This design is inherited from the smartcard world where a transaction must survive a power loss, because the power source is external and depends on a card acceptance device (CAD). *CAD* is used here to refer to both types of card readers: the conventional card acceptance device for contacted I/O interfaces and the proximity coupling device (PCD) for contactless interfaces.

Execution

The format of Java Card applications is a converted applet (CAP) file, which ensures binary compatibility across Java Card platforms. A CAP file contains an executable binary representation of the classes. Once Java programming language class files that make up a package of a Java Card applet have been generated as for any other Java application, they are preprocessed by a converter tool that converts the package to a CAP file. The converter may also produce an export file.

The Java Card platform guarantees that any update to a single persistent object field or single class field will be atomic.

Because there is no class loader in Java Card, the VM interprets the code installed on the platform by an installer on the platform. There are similarities between this process and a linking process: export files are used both to get information about packages that are imported by an application before conversion or to output information about a package that may be used by an application later after conversion. As an example, Java Card API packages are referenced in an export file used as input at the time that applet code is converted into a CAP file.

The installer is a runtime mechanism to download and install CAP files. The installer receives the executable binary from a CAD installation program, writes the binary into the smartcard memory, links it with the other classes on the card, and creates and initializes any data structures used internally by the Java Card RE.

A runtime verifier is not required by the Java Card VM specification. The verification of the bytecode of a CAP file is performed off-VM by a verifier tool after the applet has been converted into a CAP file and before the installation of the corresponding applet. This process implies that the path in between the verification of the applet and its installation is secure: the integrity and authenticity (usually also the confidentiality) of the CAP file installed on the Java Card platform must be guaranteed. This process does not prevent, in any way, performing some additional verification at runtime, but that additional action is implementation-dependent and not mandated by the specification.

Figure 3 illustrates the development and deployment model for an applet.

Atomicity and Transactions

The Java Card platform guarantees that any update to a single persistent object field or single class field will be atomic. In addition, the Java Card platform provides single-component

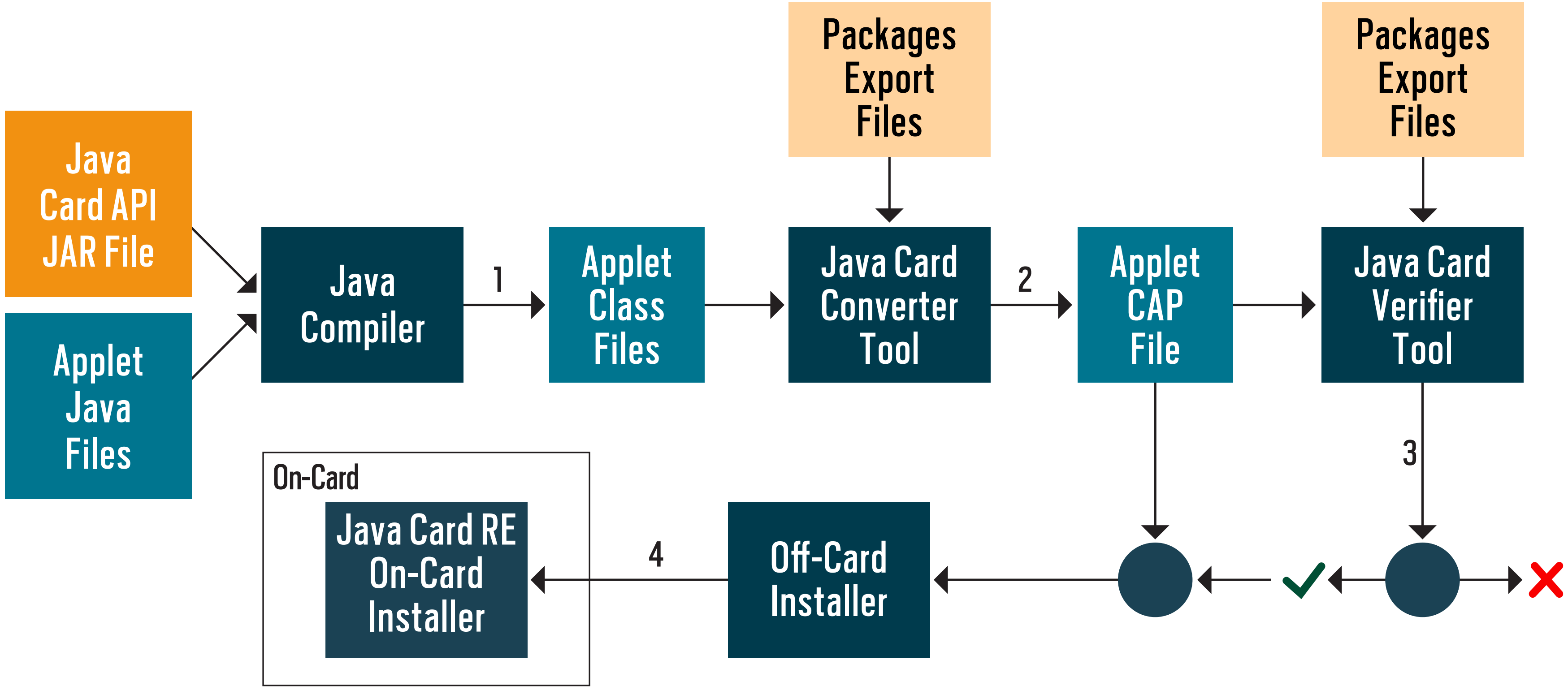


Figure 3. The applet development and deployment model

atomicity for persistent arrays. That is, if a smartcard loses power during the update of a data element (a field in an object, class, or component of an array) that should be preserved across CAD sessions, that data element will be restored to its previous value. Some methods also guarantee atomicity for block updates of multiple data elements. For example, the atomicity of the `Util.arrayCopy` method guarantees that all bytes are correctly copied; otherwise, the destination array is restored to its previous byte values. An applet might not require atomicity for array updates. The `Util.arrayCopyNonAtomic` method is provided for this.

An applet might need to atomically update several different fields or array components in several different objects. Either all updates take place correctly and consistently or else all fields and components are restored to their previous values. The Java Card platform supports a transactional model in which an applet can designate the beginning of an atomic set of updates with a call to the `JCSystem.beginTransaction` method. Each object update after this point in the code is conditionally updated. The field or array component appears to be updated (reading the field or array component back yields its latest conditional value), but the update is not yet committed. When the applet calls `JCSystem.commitTransaction`, all conditional updates are committed to persistent storage. If power is lost or if some other system failure occurs prior to the completion of `JCSystem.commitTransaction`, all conditionally updated fields or array components are restored to their previous values. If the applet encounters an internal problem or decides to cancel the transaction, it can programmatically undo conditional updates by calling `JCSystem.abortTransaction`.

The Java Card Forum is working to bring to market a new version of the Java Card specification that will address the IoT market and new secure element form factors.

Applet Isolation and Firewall

A Java Card *context* is a protected object space associated with each applet package and Java Card RE. All objects owned by an applet belong to the context of the applet's package.

Any implementation of the Java Card RE supports isolation of contexts and applets. *Isolation* means that one applet cannot access the fields or objects of an applet in another context unless the other applet explicitly provides an interface for access.

A critical security feature of Java Card is the applet firewall. This technology is runtime-enforced protection and is separate from the Java programming language protections, which still apply to Java Card applets. They ensure that strong typing and protection attributes are enforced.

Applet firewalls are always enforced in the Java Card VM. They enable the VM to automatically perform additional security checks at runtime.

In addition, the Java Card RE maintains its own Java Card RE context. This context is much like the context of an applet, but it has special system privileges so that it can perform operations that are denied to applets contexts. Figure 4 illustrates security in a multiapplet architecture.

Isolation of applets is an important security feature, but it requires a mechanism to allow applets to share objects in situations where there is a need to interoperate. The Java Card RE

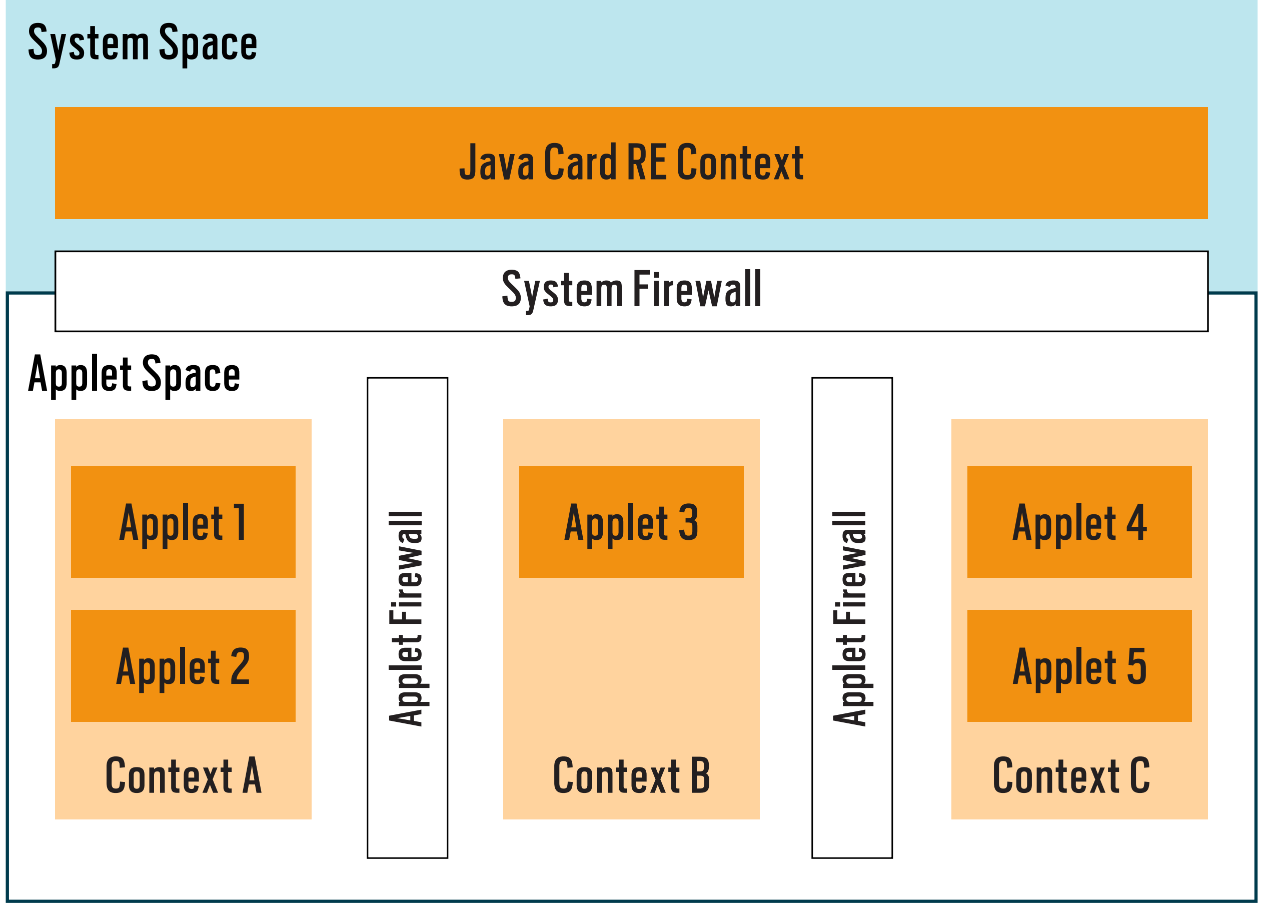


Figure 4. The security in multiapplet Java Card implementations

allows such sharing using the concept of *shareable interface objects*. These objects provide the only way an applet can make its objects available for use by other applets. For more information about using shareable interface objects, see the description of the interface `javacard.framework.Shareable`.

The Client/Server APDU Model

The interface for communicating with a Java Card applet is a packet mechanism: application protocol data units (APDUs). Related specifications are contained in the ISO 7816 Part 1 through Part 6 documents. For the purpose of developing Java Card applets, the most relevant document is ISO 00207816-4 (application level). Lower levels, such as physical levels, may vary—for

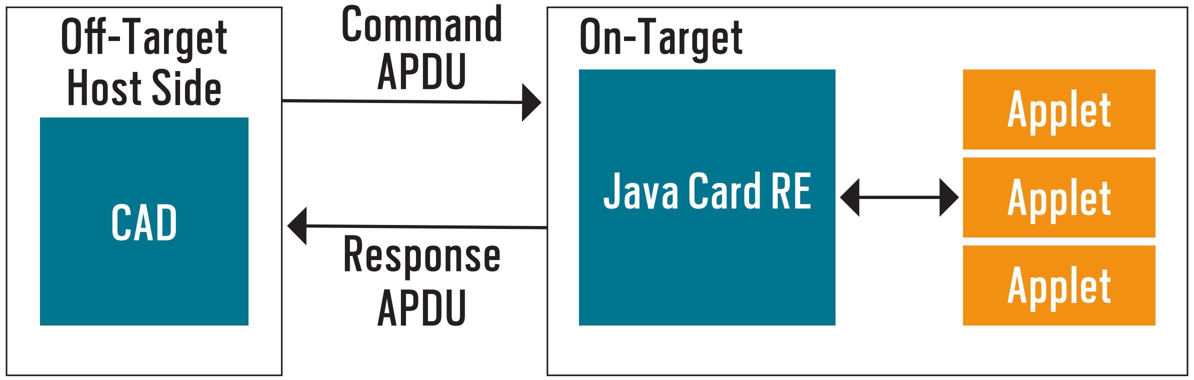


Figure 5. Java Card communication with outside devices

example, serial port, I2C, and SPI for contacted I/O interfaces and ISO 1443 or SWP for contactless interfaces.

The communication model is a command-response model where a Java Card applet acts as a server receiving requests from a client application running within the CAD (see Figure 5). The Java Card VM processes one command at a time (there is no thread support), but the runtime can manage different sessions with a given applet and different applets at the same time.

Defining the protocol supported by an applet entails defining the APDUs to process. This is one of the first steps (if not *the* first step) to developing an applet.

The Application Model

All Java Card applications must extend the `javacard.framework.Applet` class. The following are the typical methods to implement.

The applet constructor is invoked only once by the `install()` method. It serves to allocate objects that will be used during the entire lifetime of the applet to ensure that the applet will not lack memory.

The `install()` static method is invoked by the Java Card RE during the applet installation process to create an instance of the applet. The applet should perform any necessary initializations and must call one of the `register()` methods successfully to complete the installation process. The `register()` method specifies the applet identifier (AID), as defined in ISO 7816-5, of the applet to be used to select the applet later.

The `select()` method is invoked by the RE to inform an applet that it has been selected by a CAD application willing to establish a session with it. It returns `true` to indicate that the applet is ready to receive APDU commands.

The `deselect()` method is invoked by the RE to inform a selected applet that it has been deselected. The applet might then perform session cleanup.

The `process()` method is invoked by the Java Card RE to process an incoming APDU command. The applet is expected to perform the requested action and provide response data, if there is any, in return.

Conclusion

This article provided an overview of how Java Card works today. As you can see, the programming is straightforward. It's the security aspects that require experience and skill. Oracle and its partners in the Java Card Forum are working to bring to market a new version of the Java Card specification, version 3.1, which will introduce functionality to address the IoT market and new secure element form factors. It is intended for release in 2018 and will be the focus of a follow-up article in *Java Magazine*. `</article>`

Nicolas Ponsini is a security solutions architect at Oracle. He is an expert in security, cryptography, IoT, and Trusted Execution Environment and holds nine patents in related areas.

Florian Tournier is senior director, development, at Oracle. He heads a team that builds and delivers Java products for small devices and IoT security. He joined Oracle in 2010 as part of the Sun Microsystems acquisition, where he was heading Java ME and Java Card product management. Tournier holds a master's degree in management from HEC Paris and a computer science degree from the Ecole Centrale in Lyons.

SIMON ROBERTS

Quiz Yourself

Intermediate and advanced test questions

Before entering into my usual preamble about the quiz questions, let me share some corrections and refinements. The first, from Chris Noë, is that contrary to what I mentioned in the answer for question 3 in the July/August issue, the `newWatchService` method in the `FileSystem` class is not static. He's right; it is a factory, but it's not static—it's an instance method.

Another reader, Robert Filman, noted that in my answer for question 4 of the same issue, regarding how compilers might reorder the code, I suggested that reordering two particular methods would not change the outcome of the code. I had created the methods with no arguments, but I failed to state the idea that was in my head, which was that the methods should have no side effects. If they have side effects, my assertion about reordering would be unsound—indeed, compilers are unlikely to reorder method calls precisely because it's too difficult for them to know whether the methods have side effects. Compilers typically reorder things they can see in their entirety—for example, two arithmetic expressions.

The discussion was intended to be illustrative, and it's hard at times to strike the right balance between filling in all the details and getting to the point. (It's possible that you think it takes me too long to get to the point anyway!)

I very much welcome readers to write to me with concerns or questions. Everyone gains by sharing in the discussion or correction. Email me care of *Java Magazine* at javamag_us@oracle.com.

If you're a regular reader of this quiz, you know that these questions simulate the level of difficulty of two different certification tests. Those marked "intermediate" correspond to questions from the Oracle Certified Associate exam, which contains questions for a preliminary level of certification. Questions marked "advanced" come from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise.

Let me re-emphasize that these questions rely on Java 8. I'll begin covering Java 9 in future columns, of course, and I will make that transition quite clear when it occurs.

Question 1 (intermediate). Given this code:

```
int a = 012; // line n1
int b = 12;
int c = Integer.parseInt("012", 10);
```

and these statements:

1. `a == b`
2. `b == c`
3. `a == c`
4. **Line n1 causes a compilation error.**

Which is true? Choose one.

- **A.** 4 only
- **B.** 1, 2, and 3
- **C.** 1 only
- **D.** 2 only
- **E.** 3 only

Question 2 (intermediate). Given this fragment (shown with line numbers):

```
14: long i1 = 1234567890123456789;
15: short s1 = 99, s2 = 100, s3 = s1 + s2;
16: float pi = 3.14;
17: short s = 199;
```

Which is true? Choose one.

- **A.** Lines 14 and 16 are correct, but lines 15 and 17 cause compilation to fail.
- **B.** Lines 14 and 15 are correct, but lines 16 and 17 cause compilation to fail.

C. Lines 15 and 17 are correct, but lines 14 and 16 cause compilation to fail.
D. Line 17 is correct, but lines 14, 15, and 16 cause compilation to fail.
E. Line 14 is correct, but lines 15, 16, and 17 cause compilation to fail.

Question 3 (advanced). Given that the following declaration has been properly initialized to refer to a `List` implementation that contains multiple `String` objects, and that your intention is to remove from the list all strings that start with *:

```
List<String> ls;
```

Which of the following are true? Choose two.

A. The items can properly be removed by using this code:

```
for (String s : ls) {
  if (s.startsWith("*")) ls.remove(s);
}
```

B. The items can properly be removed by using this code:

```
ls.forEach(s->{if (s.startsWith("*")) ls.remove(s);});
```

C. The items can properly be removed by using this code:

```
ListIterator<String> lis = ls.listIterator();
while (lis.hasNext()) {
  if (lis.next().startsWith("*")) lis.remove();
}
```

D. The items can properly be removed by using this code:

```
int last = ls.size();
for (int idx = 0; idx < last; idx++) {
  if (ls.get(idx).startsWith("*")) {
    ls.remove(idx);
  }
}
```

E. Given the information available, it's not possible to guarantee that the removal will succeed.

Question 4 (advanced). Given the following code:

```
Optional
 .<String>ofNullable(null)
 .flatMap(x->x != null ? Optional.of("A") : Optional.of("B"))
 .map(String::toLowerCase)
 .ifPresent(System.out::println);
```

Which is true? Choose one.

A. The output is `a`.
B. The output is `b`.
C. Replacing the `flatMap` call with `.orElseGet(()->Optional.of("C"))` results in the output `c`.
D. The code runs but produces no output.
E. Replacing the call to `flatMap` with `.map(x->x != null ? "A" : "B")` results in the output `b`.

Question 1. The correct answer is option D. This question investigates an obscure corner of Java's numeric literal syntax. For reasons that are largely historical, Java supports the octal (base-8) number system. In this system, you would count to 10 as "1, 2, 3, 4, 5, 6, 7, 10, 11, 12." The system uses only eight digits (0 through 7). After reaching 7, it sets the first column to 0 and increments the next column—going from 7 to 10 (pronounced "one-zero," not "ten")—in the same way that the familiar base-10 system goes from 9 to 10. However, a critical difference is

that the digits 10 in octal represent one batch of eight and no extra units, whereas 10 in base-10 represents one batch of ten and no extra units.

Java source code indicates that the programmer is using octal by putting a leading zero on the digit stream. In other words, the literal assignment in `int a = 012` is actually assigning a value of 10, not 12. That's one batch of eight and two extra units. Importantly, however, the compiler is completely happy with this format (see *Java Language Specification* section 3.10.1). Therefore, option A, which suggests this assignment results in a compilation failure, is incorrect.

The `parseInt` method takes an optional second parameter, which is the *radix* or, more simply, the base of the representation. The single-argument overload of this method always assumes a base-10 conversion. The two-argument overload used here will convert the text to base-10, because I specifically provided a second argument. Therefore, the value of the variable `c` will actually be 12.

Given that variable `a` contains the value 10 and both `b` and `c` contain 12, it's clear that the only true statement is `b == c`, which tells you that option D is the correct answer.

Question 2. The correct answer is option D. This question investigates Java's casting and initialization rules. These sometimes seem a bit inconsistent, but there's actually sound logic to them.

On line 14, there's an attempt to assign a literal value, 1234567890123456789, to a variable of type long. The variable is big enough to hold a number that large, but unfortunately, the expression on the right side of the assignment is taken to be an int expression. An int is not big enough to hold that (it is limited to a little over +/- 2 x 109), so this line fails to compile.

You could fix this by appending the single letter `L` to the numeric literal, as shown below. (Although it's somewhat unusual for Java, the `L` is not case-sensitive in this situation, but it's generally a bad idea to use lowercase, because it looks too much like a 1 in many fonts.)

```
14: long i1 = 1234567890123456789L;
```

That suffix tells the compiler it's building a long literal, and then things work as expected.

Notice also that adding a cast to the literal won't work. An integer expression can be cast to a long expression (although it's not necessary to do so), but that doesn't affect the type of the literal that forms the expression. So, line 14 fails to compile.

Line 15 attempts to declare and initialize three short values. The range of a short in Java is -32,768 to +32,767 (that is the range of a 16-bit, two's-complement binary number). Therefore, it's certainly possible to fit 100, 99, and even 199 into the short. In fact, the first two assignments work successfully; the compiler works out that the literal expressions 100 and 99 (which you saw above are of int type) will fit without loss of precision into the variables s1 and s2. However, the assignment of s3 does not work. The problem here is that the compiler sees an expression involving two short values and the plus operator. All arithmetic is performed using at least int-sized values and, therefore, the result of the add operation is an int-sized value, not a short value. At that point, the compiler complains, because it cannot safely take an unknown int value and assign it to a short unless you cast that value. (Of course, the cast isn't "safe," but you'd be accepting responsibility for any problems.) Because of this, line 15 also fails to compile.

The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

One way to make line 15 compile would be to add a cast:

```
15: short s1 = 99, s2 = 100, s3 = (short)(s1 + s2);
```

Interestingly, however, you could change the code so that the declarations of s1 and s2 are final. If you do that, the compiler recognizes that the value of the expression being assigned to s3 must be 199 (s1 + s2 becomes a constant expression at compile time), and the compiler is once again willing to perform the assignment without complaint.

Line 16 might look simple enough, but it, too, fails to compile. The reason is that the literal value 3.14 is a double expression, and attempting to assign this to a float value will fail. The value isn't too large for a float, but it will likely lose precision, and that's sufficient reason for the compiler to object. In this case, you can add the F suffix to the literal, ensuring that the literal expression is of type float from the beginning. But unlike with the long value, you can successfully add a cast in front of this one:

```
16: float pi = (float)3.14;
```

You might think this inconsistent given that line 14 could not be fixed by adding a cast. The difference is that in line 14, the literal expression was not valid, so an attempt to cast it would be too late to fix it. However, in line 16, the literal is valid, so a cast can be applied to it, and in so doing, you overcome the problem of assigning the value to the float-sized variable. It's probably fair to suggest that the F suffix, making the literal a float-type expression, is the preferred style.

Line 17 compiles successfully. As I mentioned earlier, assigning a constant expression to a variable that's large enough for the actual value being assigned is permitted, even if the size of the type of the constant expression (which is int) is too large for the type of the variable being assigned. Given this, you can see that only line 17 compiles and option D is the correct answer.

Question 3. The correct answers are options C and E. This question investigates the rules related to mutation of collections, particularly during iteration. In Java 8, there are several ways to iterate over a list. These include using the enhanced for loop, the forEach method, the ListIterator (and the simple Iterator that's available on any Iterable), and a stream. As a general rule, any of these "built-in" iteration mechanisms carry warnings that modifying the data structure during the iteration is unsafe and likely (but not guaranteed) to result in a ConcurrentModificationException being thrown.

One safe approach could be to use an external iteration, such as the one shown in option D. However, the one shown in option D is fatally flawed, because it iterates through to a limiting index (the last variable), which is extracted at the start of the iteration. Therefore, if any items are removed, this code is guaranteed to iterate beyond the end of the list, and so option D is incorrect.

An Optional lets you pass a real object that wraps another object that might or might not exist.

The other generally safe approach is that the Iterator and ListIterator interfaces define a remove method. These methods are intended to be able to remove the most recently seen element from the structure being iterated. The code in option C correctly uses this feature and, therefore, option C is correct.

However, although the remove method and the style of its use in option C are generally correct, the remove method is an optional operation, because the underlying list might be immutable. This means that option E is also correct.

In fact, it's the API documentation for the remove method of Iterator that tells you any other kind of modification is unsafe. It says, "The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method."

Of course, this notation doesn't entirely explain why options A and B should be unsafe. The answer lies in the underlying mechanisms of the enhanced for loop and the forEach method. As you've probably guessed by now, they use the Iterator in their implementations. *Java Language Specification* section 14.14.2 describes how the enhanced for construction is translated into a regular for loop and the Iterator is extracted and used. The API documentation for the Iterable forEach method similarly describes the likely implementation of the forEach behavior in terms of the Iterable. Given this, it's clear that options A and B must also be incorrect.

Question 4. The correct answer is option D. This question investigates the essential behavior of an Optional object. An Optional allows you to avoid passing null pointers around and the associated need to test for them frequently. Instead an Optional lets you pass a real object that wraps another object that might or might not exist. The Optional allows you to avoid writing code like the following:

```
String s = operationThatMightReturnNull();
```

```
if (s != null) {
    s = s.toLowerCase();
}
```

Instead, an Optional allows you to perform operations on the data that it wraps, creating a new Optional as a result, but if the original Optional wrapped a null, the transformation is simply skipped, thereby avoiding the clutter of checking for null in the caller. So, the above code would be replaced with this:

```
Optional<String> os = operationThatReturnsOptionalString()
    .map(String::toLowerCase);
```

In this, the operationThatMightReturnNull has been rewritten to return an Optional directly. It didn't have to be; you could wrap it using Optional.ofNullable(operationThatMightReturnNull()), but it's cleaner this way and reflects a more complete adoption of the Optional into the software's design.

The important thing is that if the initial function call returns an empty Optional—that is, an Optional that wraps a nonexistent object—the operation specified in the map call is never called, and the result of the entire chain is an empty Optional.

Side note: Although as a rule the map call and similar operations return a new Optional object, the class's implementation appears to be smart about representing emptiness; it generally reuses the same underlying instance for every empty Optional. That's not guaranteed behavior (the API documentation warns against that assumption), but it's a smart way to save memory and has no other consequences, unless you do some strange and improbable things with == operations without knowing about this singleton design.

The operation flatMap is essentially the same as a map, except that it's used in situations where the function that is provided as an argument returns the result of the flatMap directly (that result is an Optional). Contrast this with the map operation, where the supplied function returns data that will be wrapped by the map operation into an Optional that will be returned by map. In other words, with a map operation, the supplied function creates the data—which might be null—and leaves it to the map operation to wrap that in an Optional. With flatMap, the supplied function takes responsibility for providing an Optional directly.

And now you have the essence of this question. Because the wrapped value in the question is, in fact, null, the flatMap transformation is never executed. The empty Optional that is returned from the flatMap is then used to invoke the map operation. Therefore, that map operation also skips executing its transformation, for the same reason. Finally, the ifPresent method recognizes that the value is not present, so it does not invoke the println behavior in the Consumer that is the argument to the ifPresent method. As a result, the code generates no output, and the correct answer is option D.

For the reasons just outlined, option E, which suggests replacing the flatMap with a functionally equivalent map operation, would not change the outcome. Therefore, option E is incorrect.

And given that the call to flatMap is a distraction, it's clear that both options A and B are incorrect.

So, what about option C? This looks as if it would see the empty state of the object on which it's invoked and return a nonempty value. It would, but the problem here is that the return type of the orElseGet must be the content type of the Optional on which it's invoked. Therefore, it would need to return a String to be compilable. It doesn't, so it won't compile, and option C must be incorrect. However, if it did return a String instead of an Optional<String>, you'd still have a problem, because then you'd be attempting to call the map operation on a String, and of course, map is not defined as a String operation; it's an operation in the Optional class. </article>

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@omeda.com, who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A-3133, Redwood Shores, CA 94065, USA.

☛ World's shortest subscription form

☛ Download area for code and other items

☛ *Java Magazine* in Japanese